FACULTAT D'INFORMÀTICA DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Desarrollo de un modelo de conexión sin servidor para videojuegos multijugador online

Director: Alejandro Pajuelo González
Codirector: Javier Verdú Mulá
Autor: Álvaro Mateo Álvarez

FIB - Grado en Ingeniería Informática
Memoria Trabajo Final de Grado

Cuadrimestre otoño 2016/2017

Datos del proyecto

Título del proyecto:	Desarrollo de un modelo de conexión sin servidor para videojuegos multijugador online
Nombre del estudiante:	Álvaro Mateo Álvarez
Director:	Alejandro Pajuelo González
Codirector:	Javier Verdú Mulá
Titulación:	Grado en Ingeniería Informática
Especialidad:	Tecnologías de la Información
Centro de estudios:	Facultat d'Informàtica de Barcelona
Universidad:	Universitat Politècnica de Catalunya

Miembros del tribunal

Presidente:	Jordi Fornes De Juan
Vocal:	Silvia Llorente Viejo
Vocal 2:	Juan Carlos Cruellas Ibarz
Vocal Suplente:	Josep Llosa Espuny

Cualificaciones

Cualificación numérica:	
Cualificación descriptiva:	
Fecha:	

Agradecimientos

En primer lugar, querría agradecer a mi novia todos los ánimos y la ayuda que me ha proporcionado, encargándose de todo mientras yo estaba ocupado programando o escribiendo.

También querría agradecer a mi familia y amigos sus constantes palabras de apoyo y ánimo que me han ayudado a salir de momentos en los que creía que no conseguiría acabar el proyecto.

También quiero dar las gracias a los directores del trabajo, Alejandro y Javier. Ellos han resuelto todas las dudas que me han surgido durante el proyecto y además me han dado muy buenos consejos que me han ayudado a descubrir hacia dónde debía dirigirme cuando no sabía cómo continuar.

Finalmente, agradecer a Netcentric, la empresa en la que trabajo, la oportunidad de trabajar con un horario que se adaptase a mis necesidades para poder acabar este proyecto. También quiero dar las gracias a mis compañeros de trabajo, los cuales se han mostrado siempre dispuestos a ayudar en lo que pudiesen.

Resumen

Los videojuegos son uno de los sectores que más ganancias genera, debido al cada vez mayor número de personas que se unen a este tipo de diversión. Esto ha hecho que el número de empresas que se dedican al desarrollo de videojuegos aumente [1].

Uno de los aspectos más importantes de los juegos hoy en día es la conectividad que ofrecen, pudiendo jugar contra otros amigos o desconocidos, o compartir los avances personales con la comunidad de jugadores. Para ello se necesitan servidores, que guardan la información y mantienen partidas, a los cuales los jugadores se conectan para poder disfrutar del juego en línea.

Este proyecto presenta un nuevo modelo para conectar a los distintos jugadores entre sí. Sus características principales consisten en la reducción del número de servidores necesarios para mantener conectados a los jugadores, lo que conlleva una reducción de costes para la empresa, y una mayor robustez en el sistema frente a posibles desconexiones o fallos en los servidores del juego.

Índice

Índice de tablas

Índice de imágenes

1. Introducción

El mercado de los videojuegos mueve muchísimo dinero. En el año 2016 se alcanzaron casi 100 billones de dólares en ventas [1], incluyendo juegos de todo tipo y para cualquier plataforma. Pero hay un factor que casi todos los juegos más exitosos tienen en común, independientemente de la plataforma o el tipo de juego: se trata de la posibilidad de jugar a través de Internet contra o con otros jugadores.

Pasando a analizar más concretamente el mercado de los videojuegos para PC, podemos comprobar que de los 20 juegos más vistos en www.twitch.com (una de las páginas más importantes de retransmisión de videojuegos), la mayoría tienen algún modo de juego en línea [2]. Cada juego en línea tiene unos servidores que permiten la comunicación entre los distintos jugadores, por lo que una mejora que pueda tener un impacto en los costes de mantenimiento de dichos servidores puede ser algo muy atractivo para las distintas empresas del sector.

Este proyecto, que es un Trabajo Final de Grado de la especialidad de Tecnologías de la Información de la Facultad de Informática de Barcelona (Universidad Politécnica de Cataluña), pretende estudiar la viabilidad de un nuevo modelo de conexión a través de Internet para videojuegos multijugador online mediante el cual no sea necesario disponer de servidores dedicados ejecutando el juego.

El método que los videojuegos multijugador online usan para transmitir la información a los jugadores siempre ha sido un punto crítico en cuanto al rendimiento. Hoy en día, el más usado consiste en usar un servidor que haga de repetidor de la información de todos los jugadores junto con algunas otras tareas de predicción y validación de información. Si en lugar de tener un servidor actuando como repetidor se consiguiera que los jugadores se comunicasen entre sí, entonces el número de servidores destinados a mantener la comunicación entre jugadores y las partidas activas disminuiría. No sólo eso, sino que también sería mucho más fácil soportar cambios bruscos en el número de jugadores y partidas (franjas de tiempo concretas en las que se conectan más jugadores de lo habitual) y así se reduce la probabilidad de saturación del servicio.

Menos servidores implican unos costes de inversión y de mantenimiento menores, a parte de una mayor escalabilidad si el juego resulta un éxito y el número de jugadores crece demasiado rápido. Pero no todo son ventajas, ya que al no tener servidores dedicados controlando las partidas del juego, esta responsabilidad recae sobre los propios usuarios, con lo que resulta más sencillo realizar trampas para engañar al resto de jugadores. Esta es la razón por la que las empresas de videojuegos importantes no implementan un modelo de este tipo. Pero puede ser justo lo que necesitan empresas pequeñas sin el presupuesto necesario para montar toda una infraestructura de servidores.

Un posible campo (aunque no muy lucrativo) en el que tendría aplicación este tipo de conexión es el de los videojuegos Open Source. Es ahí dónde este proyecto va a desarrollarse, para así poder valorar su funcionamiento aplicado a algún videojuego ya existente sin tener que crearlo desde cero.

1.1. Estado del arte

El método que los videojuegos usan para que dos usuarios que juegan a través de Internet se comuniquen ha ido cambiando a lo largo del tiempo, pasando por distintas fases a medida que se han ido descubriendo nuevas técnicas para mejorar el rendimiento [3].

Con los primeros videojuegos que tuvieron un modo multijugador a través de Internet, la técnica de conexión era el *peer-to-peer*, usado actualmente en programas para compartir ficheros (BitTorrent, Ares, etc.), por poner un ejemplo. Este método consiste en que todos los clientes/jugadores se conecten entre sí formando una malla completa (Figura 1). Cuando uno de los jugadores da una orden a través del ratón o teclado, la información de la acción realizada es enviada a todos los demás jugadores de la partida. Este método tiene como ventaja su bajo coste, ya que no hay que tener ningún servidor dedicado. Como inconvenientes cabe destacar el hecho de que a medida que el número de jugadores aumenta, el rendimiento del juego disminuye. Otra desventaja es que el juego se pausa a menudo si hay algún jugador con una mala conexión a Internet, ya que se necesitan los datos de todos los jugadores para poder continuar la partida.

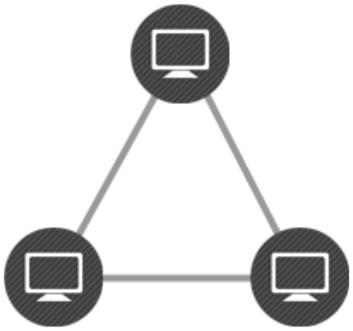

Figura 1. Modelo de conexión peer-to-peer formando malla completa

Este método *peer-to-peer*, aún se sigue usando hoy en día en juegos de estrategia en tiempo real, como por ejemplo Age of Empires [4]. Los primeros problemas de este modelo de conexión aparecieron con el primer juego de acción en primera persona: *Doom*. Este estilo de juegos, conocido como FPS (First Person Shooter) no funcionaba con el modelo *peer-to-peer*, ya que se tardaba demasiado en recibir la información de todos los jugadores y eso hacia que el juego se entrecortase. Este hecho propició la aparición del modelo de conexión que se usa hoy en día: cliente/servidor.

Con la aparición del género de videojuegos FPS se creó el modelo de conexión multijugador cliente/servidor. Este modelo es muy sencillo: hay un ordenador que actúa como servidor, y todos los jugadores/clientes se conectan a ese ordenador en lugar de comunicarse entre ellos (Figura 2).

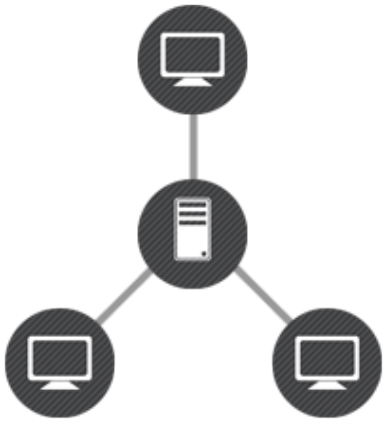

Figura 2. Modelo de conexión cliente/servidor

El servidor recibe la información de las acciones de todos los jugadores y se encarga de enviarla al resto. Dicho servidor puede ser pagado por la compañía que ofrece el juego o por los mismos jugadores, pero es un gasto a tener en cuenta.

Este nuevo modelo es capaz de solucionar los problemas que tenía el anterior utilizando nuevas técnicas de predicción de movimiento y compensación de latencia - el tiempo que tarda en llegar la información enviada al resto de jugadores. Además, el servidor realiza también funciones de prevención de trampas por parte de los jugadores, añadiendo diferentes validaciones a la información enviada. Esto no quiere decir que sean infalibles, pero si que es más complicado para los clientes saltarse las normas del juego.

Una característica muy importante de este modelo es que el servidor tiene una autoridad suprema sobre los clientes. Se encarga de ejecutar todos los cálculos pertinentes y decide en qué posición se encuentra cada jugador, si alguno ha sido alcanzado por algún proyectil y otras decisiones de este estilo. Los clientes reciben los resultados del servidor y muestran lo que este ordena [5].

1.2. Motivación

Desde que decidí que quería estudiar informática que me he visto en el futuro como desarrollador de videojuegos. Entre esto y el hecho de que no me era posible realizar el Trabajo de Fin de Grado en la empresa en la que trabajo, opté por intentar hacer algo relacionado con el desarrollo de videojuegos.

Dadas la especialidad que había escogido (Tecnologías de la Información) y el tema, estaba claro que tenía que pensar en algo que tuviera que ver con el sistema de conexión de algún videojuego. Así me puse a buscar entre las diferentes ideas para videojuegos que tenía, hasta que di con esta.

Lo que me hizo decantarme por esta opción fue la situación del mercado de videojuegos independientes. La mayoría de juegos de este tipo que conozco son juegos que no disponen de un sistema multijugador online. Creo que esto es debido a la complicación y trabajo añadido de tener que programar el sistema multijugador y al coste económico que este conlleva. Es por esto que pienso que esta idea de proyecto

tiene potencial para eliminar el segundo problema. Quizás es lo que se necesita para facilitar la apertura del mercado de los videojuegos multijugador online a estudios pequeños.

Finalmente, con este proyecto pretendía adquirir conocimientos útiles para el mercado laboral de los videojuegos para que me pudieran facilitar el acceso. También quería usar el lenguaje C++ para desarrollar el proyecto, ya que es de los más comunes para crear videojuegos y desde primero que no lo usaba, así que el proyecto me iba a servir para repasar mis conocimientos e incluso ampliarlos.

1.3. Stakeholders

Stakeholders es un término inglés popularizado por R.E. Freeman, que lo usó para referirse a “quienes son afectados o pueden ser afectados por las actividades de una empresa” [6]. Los stakeholders o principales actores de este proyecto son los siguientes:

1) Director, codirector y desarrollador. Son las personas que se van a encargar de hacer que el proyecto avance y de que se consigan cumplir los objetivos en el tiempo disponible para realizar el Trabajo Final de Grado. El director Alejandro Pajuelo González y el codirector Javier Verdú Mulá tienen la función de guiar la investigación por dónde consideren más oportuno. El desarrollador, y autor de esta memoria, se encarga de implementar el sistema y comprobar su correcto funcionamiento.

2) Empresas de videojuegos. Son las entidades que podrán sacar partido del software desarrollado en este proyecto, integrándolo en sus productos y aprovechando las ventajas que ofrece. La más importante será la reducción de los costes de compra y mantenimiento de servidores. Esto les permitirá sacar mayor beneficio. Además, es el software ideal para empresas pequeñas y emergentes, ya que muchas veces no disponen del capital necesario para poder pagar unos buenos servidores, que pueden llegar a ser muy caros dependiendo de las necesidades del juego y del número de usuarios que tenga.

1.4. Estructura del documento

En este documento se detalla el proceso seguido para elaborar el Trabajo de Fin de Grado correspondiente; desde el análisis del estado del arte y pasando por la implementación, hasta los resultados y conclusiones.

En el primer capítulo se introduce el tema de los videojuegos multijugador online y sus necesidades. También se aporta una visión general del estado del arte y una breve explicación de las motivaciones personales por las que escogí este proyecto.

En el capítulo 2 del documento se describe de qué trata exactamente el trabajo realizado. Primero se describe el problema para el cual se quiere proporcionar una posible solución y luego se definen los objetivos del trabajo, con sus correspondientes restricciones y limitaciones. También se proporciona una descripción de las tareas que se han tenido que llevar a cabo, basadas en los objetivos a conseguir, y las planificaciones temporal y económica iniciales que se hicieron.

A continuación, en el capítulo 3, se proporciona una explicación de los conocimientos previos sobre las distintas tecnologías que se han necesitado para poder desarrollar el proyecto.

En el capítulo 4 se detalla la descripción del sistema que se ha implementado. Esto comprende las características que se desean obtener como resultado, junto con una descripción de cómo es el protocolo de comunicación que sigue el sistema desarrollado.

Una vez especificado el proyecto que se va a llevar a cabo, pasamos en el capítulo 5 a dar una explicación de cómo se ha realizado la implementación de las tareas requeridas para el funcionamiento del sistema. La sección está dividida en las 3 tareas principales que se han tenido que realizar: la implementación del cliente, del servidor y la implementación para recopilar información necesaria sobre la CPU de un ordenador.

Una vez acabada la implementación, tiene lugar la evaluación del sistema, que es sobre lo que trata el capítulo 6 del documento. Primero se explica qué tipo de metodología se ha usado para evaluar el funcionamiento del sistema y luego se detalla cómo ha sido evaluado.

Seguidamente, en capítulo 7, se incluye la planificación final resultante del proyecto, para poder ver la desviación respecto de las estimaciones iniciales. Esta planificación incluye datos sobre el tiempo y los recursos económicos empleados en la realización del proyecto.

Antes de acabar se encuentra en el capítulo 8 un estudio sobre la sostenibilidad del proyecto. Esta sostenibilidad tiene en cuenta la producción del proyecto, su vida útil y sus riesgos. Para cada uno de ellos el proyecto tiene asignada una puntuación que sirve para valorar cuán sostenible es.

Finalmente, en el capítulo 9, se expone la conclusión del documento. Ahí se analiza el estado del proyecto, así como su posible uso. También se presentan posibles mejoras a realizar a continuación y una breve opinión personal sobre el Trabajo de Fin de Grado.

2. Descripción del proyecto

Teniendo en cuenta el aumento en el número de videojuegos multijugador online que se puede observar actualmente en el mercado, tanto de aplicaciones móviles como en juegos de videoconsolas o de PC, sería bueno disponer de un sistema que permitiese conectar de forma sencilla a los jugadores sin necesidad de invertir demasiado en servidores.

El problema que tienen los desarrolladores de videojuegos independientes y estudios pequeños es que no se pueden permitir desplegar unos servidores que puedan almacenar datos y partidas de muchos jugadores. Esto hace que la mayoría de desarrolladores de videojuegos independientes no creen juegos multijugador online, ya que no disponen del capital necesario para ofrecer un buen servicio. El resultado es que las grandes empresas del sector de los videojuegos son las que se reparten casi todos los beneficios de los videojuegos online.

Dada la situación desfavorable en el mercado de los videojuegos online para los estudios pequeños de videojuegos, este proyecto intenta ofrecer una posible solución para crear un hueco en el mercado para ellos. Un sistema que no necesite más que unos pocos servidores, con el correspondiente bajo capital necesario, y que sea robusto frente a fallos de red. Dicho sistema sería capaz de ofrecer un buen servicio, barato y sencillo de usar que podría proporcionar una pequeña parte de los beneficios generados por los videojuegos online a las pequeñas empresas y desarrolladores independientes.

Como es normal, no hay nada perfecto, y este sistema no es una excepción. A cambio de las ventajas que ofrece también tiene sus contrapartidas. La principal es la baja seguridad que ofrece, ya que no disponer del control sobre la ejecución de las partidas facilita la aparición de trampas para los juegos. Este es el aspecto más negativo, pero también hay que tener en cuenta que los videojuegos *indie* (independientes) no tienen un gran número de usuarios, cosa que hace menos probable la aparición de programas que falseen la información. Por tanto, la baja seguridad del sistema es un riesgo asumible.

El sistema no tiene que ser único tampoco. Se puede usar en combinación con otro tipo de modelos de conexión más seguros. Esto permitiría que los estudios que consigan éxito con su videojuego multijugador pudiesen más adelante cambiar a los modelos de conexión cliente/servidor, más convencionales en el mercado actual y que ofrecen una mayor seguridad ante posibles atacantes.

2.1. Formulación del problema

Uno de los problemas para poder desarrollar un videojuego multijugador online hoy en día es el mantenimiento de las partidas en servidores dedicados. Este es el modelo que siguen las grandes empresas, ya que permite un mayor control sobre la ejecución de las partidas.

La principal consecuencia de tener que mantener las partidas de un videojuego en servidores es el capital que se necesita. Hay que realizar una inversión inicial considerable para comprar los servidores y además estos precisan de costes adicionales de mantenimiento y energía.

El capital no es la única consecuencia de este problema. La escalabilidad de clientes del juego está limitada por el número de servidores disponibles. Esto implica que si en un momento dado la afluencia de jugadores aumenta demasiado, para mantener la calidad del servicio ofrecido se deberá aumentar el número de servidores, cosa que lleva un tiempo (hay que comprar el material, recibirlo, instalarlo…).

Otra consecuencia es la caída o saturación de los servidores. Si hay demasiadas partidas que mantener por un exceso de jugadores a los que se puede dar servicio, los servidores se saturan e incluso pueden llegar a desconectarse. En este caso, todos los clientes del juego se ven privados de poder jugar a través de Internet, ya que el sistema multijugador no funcionará. Las consecuencias de esto son desastrosas para la empresa que ha desarrollado el juego y mantiene los servidores, ya que puede perder jugadores que han quedado insatisfechos debido a la falta de un buen servicio.

2.2. Objetivos

El objetivo principal de este proyecto es diseñar un sistema de conexión para videojuegos multijudador online en el que las partidas no necesiten de un servidor central en el que ejecutarse. Por el contrario, las partidas se ejecutarán en los propios ordenadores de los clientes/jugadores. No se trata de crear un sistema de juego *peer-to-peer*, sino de un servidor que se encargue de preparar las partidas juntando a los jugadores para que luego estas pasen a ejecutarse en el ordenador de alguno de ellos.

El servidor podrá ser usado en cualquier juego. Para usar este sistema, el juego tendría que tener disponible tanto el código del cliente como el del servidor. Si se cumple esto, llamando a algunas de las funciones del sistema se recibirá información que indicará al cliente si debe actuar como servidor y esperar al resto de jugadores, o si tiene que conectarse al ordenador de otro de los jugadores, en cuyo caso habrá recibido la dirección correspondiente.

Para conseguir llevar a cabo el objetivo principal, este se puede desglosar en los siguientes puntos más específicos:

1) Búsqueda de un juego Open Source sobre el que sea posible probar el sistema.
2) Diseño del protocolo de conexión con el servidor que organiza las partidas.
3) Implementación del código del servidor.
4) Implementación del código que se ejecutará en los clientes.
5) Evaluación del sistema

Estos objetivos específicos son también la ruta que ha seguido el proyecto para llevar a cabo el desarrollo del sistema y hasta la redacción de esta memoria.

2.3. Restricciones

Los principales problemas u obstáculos que se han presentado al realizar este proyecto son los siguientes:

1) Tiempo demasiado limitado. Este suele ser un problema recurrente en todos los proyectos, ya que se suelen querer soluciones rápidas. En el caso de este Trabajo Final de Grado, en lugar de un cliente que requiera el software que se desarrolla, hay una fecha límite de entrega. El resultado es el mismo: en poco tiempo hay que hacer muchas cosas. Por eso ya se han eliminado ciertos requerimientos del alcance del proyecto (como por ejemplo la seguridad).

2) Cliente multiplataforma. El servidor se puede ejecutar sobre cualquier sistema basado en UNIX, pero no se puede decir lo mismo del cliente, ya que el juego está disponible para Macintosh, Windows y Linux. Esto ha obligado a tener que implementar las partes que se ejecutan en el cliente de 3 maneras distintas para que funcione indistintamente de la plataforma sobre la que se esté ejecutando el cliente.

3) La evaluación no es del todo significativa. A pesar de haber implementado un sistema de ACK (acuse de recibo) para los mensajes, con reenvíos en el caso de que no hayan llegado o se hayan recibido con errores, al realizar la evaluación no se tiene en cuenta. Esto es así porque no se dispone de un servidor en el que instalar el sistema para poder probarlo en condiciones normales. Debido a esto, la evaluación se ha llevado a cabo en local.

2.4. Metodología de trabajo

Para el desarrollo del proyecto se utilizará una metodología de desarrollo en cascada [7]. Se trata de un proceso de diseño secuencial usado para el desarrollo de software. El proceso pasa por las distintas fases una por una. Estas son: requerimientos, diseño, implementación, verificación y mantenimiento (Figura 3). En la fase de requerimientos se decide qué es lo que se debe hacer. Luego se diseña cómo será el sistema. A continuación, se lleva a cabo la implementación del sistema, que luego se debe probar para comprobar que todo funciona correctamente. La fase de mantenimiento no la llevaremos a cabo, ya que se sale del alcance del proyecto.

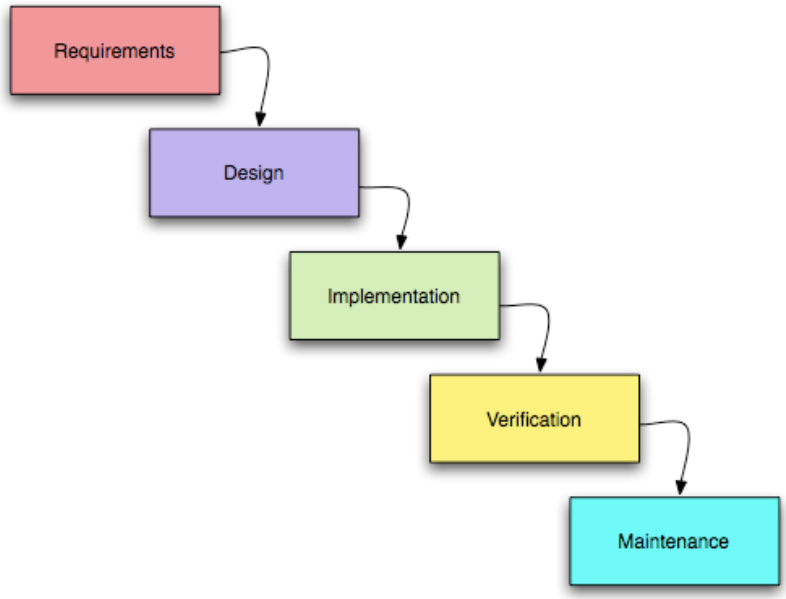

Figura 3. El proceso seguido en la metodología de trabajo en cascada

Al ser sólo un desarrollador es el método que más se adecúa al proyecto, ya que si quisiéramos usar una metodología de trabajo *Agile* necesitaríamos más trabajadores. Con la metodología *Agile*, en lugar de tener un proceso secuencial se tiene un proceso paralelo: se empieza a implementar a la vez que se hacen los diseños y se empieza a testear mientras aún se esta realizando la implementación.

Para el desarrollo del código se usará un sistema de control de versiones llamado Git. También se usará GitHub para disponer de un repositorio online al que se pueda acceder desde cualquier ordenador. Se llevará a cabo una metodología de trabajo llamada *Git Workflows* [8]. Esta consiste en la creación de una rama *feature* por cada requerimiento o detalle de implementación que haya que hacer, uniéndolas solamente cuando el código implementado en cada una compile y de esta forma asegurando que lo que haya en la rama principal (también llamada *master* o *develop*) siempre se pueda ejecutar sin problemas.

2.5. Descripción de las tareas

El proyecto se ha dividido en una serie de tareas con el propósito de cumplir los objetivos mencionados en el punto 2.1. A continuación se presenta una descripción de cada una de estas tareas:

1) *Inicio de proyecto*. En esta fase del proyecto se realizó una búsqueda sobre cuál sería el objetivo del proyecto. Una vez respondida esta pregunta, el siguiente paso fue recaudar toda la información posible y necesaria para llevar a cabo el proyecto; por ejemplo, tecnologías que se usarían y el estado del arte de lo que se ha desarrollado.

2) *Gestión del proyecto*. Esta fue una fase de descripción y organización, básicamente. Se dejaron bien definidos y especificados mediante documentos el alcance, la planificación, los costes y cualquier otra información necesaria para la realización del proyecto.

3) *Implementaciones*. Hay dos fases de implementación: la del servidor y la del modelo de conexión. En cada una se llevó a cabo un análisis de lo que tiene que ser capaz de hacer el software, para luego poder diseñar una solución que fuese capaz de resolver el problema al que nos enfrentábamos. Una vez hecho el diseño, el último paso de la fase consistía en escribir todo el código necesario para poner en funcionamiento las características o requerimientos.

4) *Validación*. En esta fase se ejecuta y se prueba el sistema desarrollado. Hay que realizar distintas pruebas para poder asegurar el funcionamiento correcto y esperado del producto.

5) *Documentación y presentación*. Esta última tarea del proyecto consiste en la redacción de esta memoría, así como la documentación del código que se ha desarrollado para que pueda ser entendido con más facilidad.

2.6. Planificación temporal

El proyecto tenía una duración inicial estimada de 4 meses, empezando el 15 de Febrero para acabar el 15 de junio. Debido a complicaciones que han surgido, que serán analizadas en el capítulo 7, se ha tenido que aplazar la fecha de entrega hasta el turno de octubre. Durante este tiempo se han ido realizando las tareas descritas anteriormente, a las cuales se les había asignado una estimación inicial del tiempo que llevaría desarrollarlas.

Estimación de tareas

Tarea	Responsable	Horas
Inicio de proyecto		**60**
1. Búsqueda y puesta en marcha	Jefe de Proyecto	10
2. Aprendizaje	Desarrollador	50
Gestión del proyecto		**70**
1. Definición del alcance	Jefe de Proyecto	20
2. Planificación temporal	Jefe de Proyecto	10
3. Gestión económica y sostenible	Jefe de Proyecto	10
4. Presentación preliminar	Jefe de Proyecto	10
5. Pliego de condiciones	Jefe de Proyecto	5
6. Presentación oral y documento final	Jefe de Proyecto	15
Implementación del servidor		**100**
1. Análisis de requisitos	Analista	10
2. Diseño	Diseñador	20
3. Implementación	Desarrollador	70
Implementación del modelo de conexión		**100**
1. Análisis de requisitos	Analista	10
2. Diseño de conexión y protocolo de comunicación	Diseñador + Desarrollador	20
3. Implementación	Desarrollador	70
Validación		**20**
1. Creación de pruebas	Ingeniero QA	10
2. Realización de tests	Ingeniero QA	10

Medición de rendimiento		**40**
1. Diseño	Ingeniero QA	10
2. Implementación	Ingeniero QA + Desarrollador	20
3. Medición	Ingeniero QA	10
Documentación y presentación		**60**
1. Redacción de la Memoria	Jefe de Proyecto	50
2. Preparación de la defensa	Jefe de Proyecto	10
Total		**450**

Tabla 1. Estimación inicial en horas de las distintas tareas del proyecto

Al principio se consideró dedicar tiempo a tomar medidas de rendimiento para comprobar que el juego funcionase igual de bien usando este nuevo sistema. Más adelante se decidió que no tenía sentido realizar esta tarea debido a que los valores serían dependientes del ordenador que hiciese de servidor, que es diferente en cada ocasión, dependiendo de que ordenadores tengan los jugadores que haya en la partida.

A continuación se puede observar el diagrama de Gantt (Figura 4) [9] resultante de la anterior estimación inicial de tiempo para el proyecto.

	Nombre	Duración	Inicio	Fin
1	1. Inicio del proyecto	11d	15/02/2016	29/02/2016
2	1.1. Búsqueda y puesta en marcha	2d	15/02/2016	16/02/2016
3	1.2. Aprendizaje	9d	17/02/2016	29/02/2016
4	2. Gestión del proyecto	30d	01/03/2016	11/04/2016
5	2.1. Definición del alcance	5d	01/03/2016	07/03/2016
6	2.2. Planificación temporal	5d	08/03/2016	14/03/2016
7	2.3. Gestión económica y sostenible	5d	15/03/2016	21/03/2016
8	2.4. Presentación preliminar	5d	22/03/2016	28/03/2016
9	2.5. Pliego de condiciones	5d	29/03/2016	04/04/2016
10	2.6. Presentación oral y documento final	5d	05/04/2016	11/04/2016
11	3. Implementación del servidor	15d	12/04/2016	02/05/2016
12	3.1. Análisis de requisitos	1d	12/04/2016	12/04/2016
13	3.2. Diseño	2d	13/04/2016	14/04/2016
14	3.3. Implementación	12d	15/04/2016	02/05/2016
15	4. Implementación del modelo de conexión	15d	03/05/2016	23/05/2016
16	4.1. Análisis de requisitos	1d	03/05/2016	03/05/2016
17	4.2. Diseño de conexión y protocolo de comunicación	2d	04/05/2016	05/05/2016
18	4.3. Implementación	12d	06/05/2016	23/05/2016
19	5. Validación	5d	24/05/2016	30/05/2016
20	5.1. Creación de pruebas	3d	24/05/2016	26/05/2016
21	5.2. Realización de tests	2d	27/05/2016	30/05/2016
22	6. Medición de rendimiento	5d	31/05/2016	06/06/2016
23	6.1. Diseño	1d	31/05/2016	31/05/2016
24	6.2. Implementación	3d	01/06/2016	03/06/2016
25	6.3. Medición	1d	06/06/2016	06/06/2016
26	7. Documentación y presentación	7d	07/06/2016	15/06/2016
27	7.1. Redacción de la Memoria	5d	07/06/2016	13/06/2016
28	7.2. Preparación de la defensa	2d	14/06/2016	15/06/2016

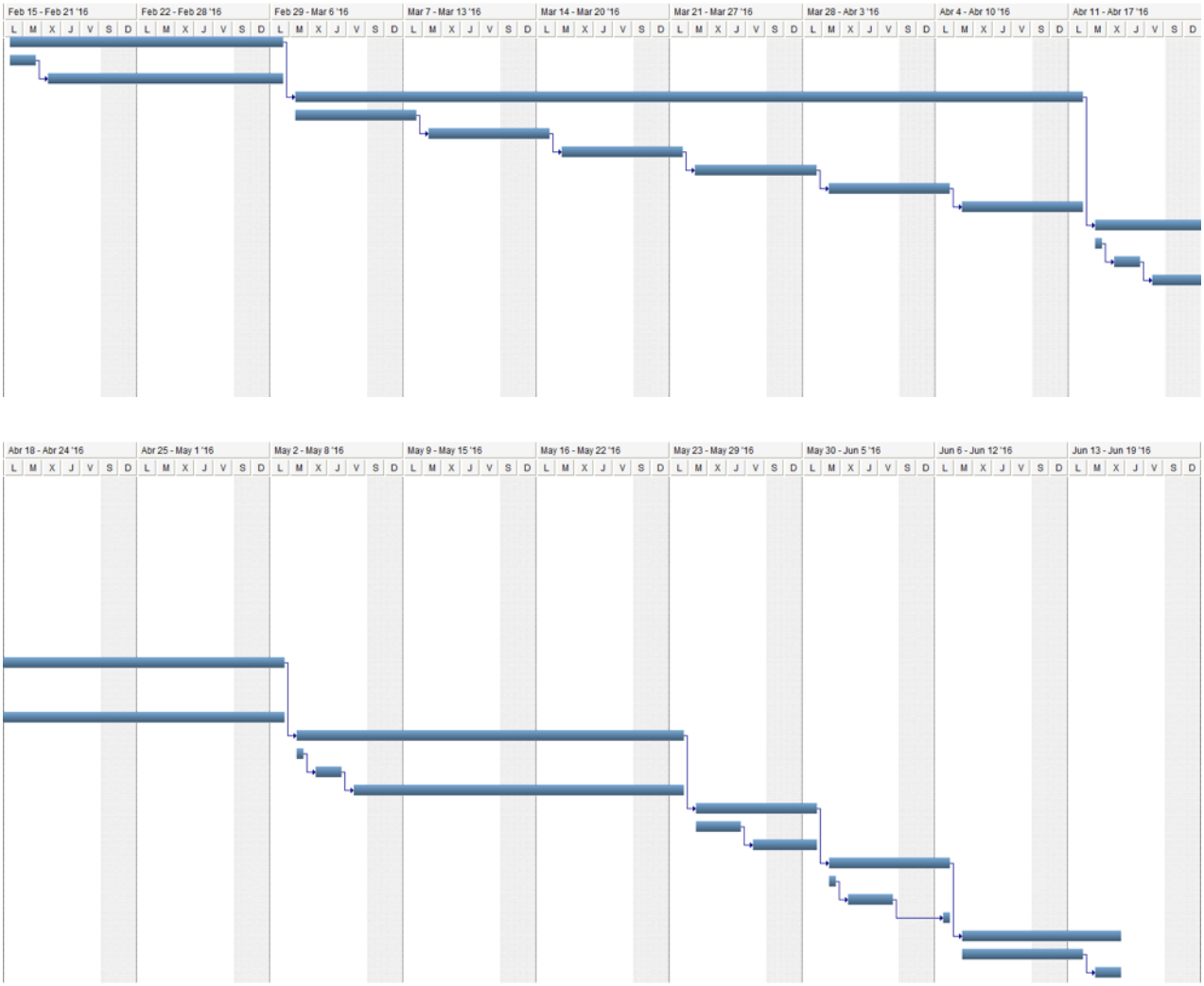

Figura 4. Diagrama de Gantt de la estimación inicial de tareas del proyecto

2.7. Planificación económica

En este apartado voy a detallar la estimación inicial de costes que se hizo para el proyecto. Para ello voy a tener en cuenta los siguientes detalles. No consideraré costes de actividad relativos al volumen de producción (ya que no se está produciendo ningún número específico de unidades, sino un servicio de software). Por tanto, todos los costes serán considerados de estructura. Para realizar la gestión de gastos se considerará la siguiente clasificación de costes: recursos humanos, recursos materiales (hardware y software) y gastos generales. A parte, también se deben tener en cuenta los costes de contingencia y de imprevistos.

Costes de recursos humanos

Los costes de recursos humanos son sencillos de calcular, ya que solamente se dispone de un trabajador que se encarga de llevar a cabo todas las tareas del proyecto. Dicho trabajador hará las funciones de jefe de proyecto, analista, diseñador, desarrollador e ingeniero QA. Al ser aún un estudiante su salario será de 8 €/hora y será el mismo para todas las tareas realizadas.

Actividad	Horas estimadas	Recursos	Salario (€/h)	Coste (€)
Búsqueda y puesta en marcha	10	Jefe de proyecto	20	200
Aprendizaje	50	Desarrollador	8	400
Inicio de proyecto	60			600
Definición del alcance	20	Jefe de proyecto	20	400
Planificación temporal	10	Jefe de proyecto	20	200
Gestión económica y sostenible	10	Jefe de proyecto	20	200
Presentación preliminar	10	Jefe de proyecto	20	200
Pliego de condiciones	5	Jefe de proyecto	20	100
Presentación oral y documento final	15	Jefe de proyecto	20	300
Gestión del proyecto	70			1400
Análisis de requisitos	10	Analista	13	130
Diseño	20	Diseñador	15	300
Implementación	70	Desarrollador	8	560
Implementación del servidor	100			990
Análisis de requisitos	10	Analista	13	130
Diseño de conexión y protocolo de comunicación	20	Diseñador	15	300
Implementación	70	Desarrollador	8	560
Implementación del modelo de conexión	100			990
Creación de pruebas	10	Ingeniero QA	15	150
Realización de tests	10	Ingeniero QA	15	150
Validación	20			300
Diseño	10	Ingeniero QA	15	150

Actividad	Horas estimadas	Recursos	Salario (€/h)	Coste (€)
Implementación	20	Desarrollador	8	160
Medición	10	Ingeniero QA	15	150
Medición de rendimiento	40			460
Redacción de la memoria	50	Jefe de proyecto	20	1000
Preparación de la defensa	10	Jefe de proyecto	20	200
Documentación y presentación	60			1200
Total	**450**			**5940**

Tabla 2. Costes directos por actividad según el diagrama de Gantt

Rol	Horas estimadas	Salario (€/h)	Coste estimado
Jefe de proyecto	140	20	2800
Analista	20	13	260
Diseñador	40	15	600
Desarrollador	210	8	1680
Ingeniero QA	40	15	600
Total	**450**		**5940**

Tabla 3. Costes totales correspondientes a los recursos humanos del proyecto

La Tabla 2 contiene detalladamente los costes directos por actividad del trabajador, siguiendo la planificación temporal de la sección 2.5. Esta tabla permite ver en que se emplea exactamente el dinero del proyecto destinado a recursos humanos. Por el contrario, en la Tabla 3 se puede ver el coste según el rol del trabajador [10], dando como resultado una versión más resumida de los costes.

Costes de recursos materiales

Estos gastos son considerados costes indirectos, ya que no dependen del proyecto que se está realizando, sino que son gastos que se tienen independientemente de si se está desarrollando algo o no. En cuanto a *hardware* sólo se necesita el MacBook Pro de 13 pulgadas con pantalla de retina que se usa para programar y realizar la evaluación de los resultados. De *software* no tenemos ningún coste añadido, ya que casi todos los programas usados son Open Source (no cuesta dinero). Lo único

que no es Open Source son los programas de ofimática usados para redactar los documentos y realizar el soporte para presentaciones: Pages, Numbers y Keynote. Pero estos programas vienen incluidos con el MacBook, así que su coste va incluido junto con el del portátil. El coste de amortización del portátil se ha calculado teniendo en cuenta que el trabajador sólo está media jornada (coste estimado = precio / vida útil / 12 meses * duración del proyecto en meses / 2).

Finalmente, faltaría por incluir el coste de imprimir el TFG y todas sus respectivas copias. Esto sería considerado un coste directo, en lugar de uno indirecto, ya que depende de la existencia del proyecto (si no hay proyecto no hay impresiones que realizar). Visitando algunos foros de estudiantes en internet [11] parece que un precio aproximado para una impresión de este tipo ronda alrededor de los 50 euros por copia (este precio incluye impresión y encuadernación). Las copias serán 5; tres para los miembros del tribunal, una para el director y otra para el codirector.

Producto	Precio (€)	Unidades	Vida útil (años)	Coste estimado (€)
MacBook Pro 13" Pantalla Retina[10]	1700	1	4	70,83
Impresiones en papel del TFG	50	5		250
Total	**1700**			**320,83**

Tabla 4. Costes indirectos debidos a recursos materiales

Costes generales indirectos

Dentro de los costes generales se encuentran todos aquellos costes indirectos que no son ni de recursos humanos ni materiales, como por ejemplo el precio del transporte, el coste de la luz o el del alquiler de la vivienda en la que vivo (lugar que va a ser donde se va a desarrollar mayoritariamente el proyecto). Estos costes generales se ha determinado que son básicamente los que se pueden ver en la siguiente tabla (el coste estimado tiene en cuenta que la duración del proyecto serán 4 meses, aproximadamente).

Producto	Precio (€/mes)	Coste estimado (€)
Internet	63	252
Local	350	1400
Luz	40	160
Agua	15	60
Gas	20	80
T-Jove 1 zona (Familia Numerosa)	28	112
Total	**516**	**2064**

Tabla 5. Costes indirectos generales del proyecto

Costes de contingencia

Se reserva algo de dinero como seguro por si surge algún problema durante el proyecto que hace que aumente su coste real respecto al coste previsto (margen de error). La partida de contingencia que se reservará para este proyecto será del 15% de los costes directos e indirectos de recursos humanos, materiales y generales.

Producto	Porcentaje	Precio (€)	Coste (€)
Recursos humanos	15 %	5940	891
Recursos materiales	15 %	320,83	48,1245
Recursos generales	15 %	2064	309,6
Total		**8324,83**	**1248,7245**

Tabla 6. Costes de contingencia del proyecto

Imprevistos

En esta sección se tienen en cuenta los distintos imprevistos y contratiempos que pueden surgir durante el desarrollo del proyecto. En nuestro caso los principales serían:

1) Avería en el ordenador portátil. En el caso de avería habría que reparar el ordenador o sustituirlo por uno nuevo. Como el tiempo es muy ajustado y el servicio técnico tardaría más de lo que nos podemos permitir en reparar el ordenador, la opción más

buena es la de comprar uno nuevo. En este caso serían 1700 euros más que gastar, aunque la probabilidad de que esto pase es muy baja (le asignamos un 5%).

2) Retraso en la implementación del proyecto. En el caso de que nos retrasemos 15 días respecto a lo previsto, cosa bastante probable (le asignamos un 20% de probabilidad de que suceda), habría que sumar el coste de las horas extras del trabajador - 8 €/h * 5 horas/día * 15 días = 120 €.

Producto	Probabilidad	Unidades	Precio	Coste (€)
Avería ordenador	5 %	1	1700	85
Retraso implementación 15 días	20 %	75 (horas)	8 €/hora	120
Total				**205**

Tabla 7. Costes destinados a imprevistos

Presupuesto final

Para acabar, poniendo en común todo lo calculado en los apartados anteriores, se proporciona una última tabla con la información sobre el presupuesto necesario para llevar a cabo el proyecto. Al ser un proyecto Open Source no se añade ningún margen de beneficio al presupuesto, por lo que el precio que se ve es el precio real de lo que se estima que costará llevar a cabo el desarrollo de este software.

Concepto	Coste (€)
Recursos humanos	5940
Recursos materiales	320,83
Costes generales	2064
Contingencia	1248,7245
Imprevistos	205
Total	**9778,5545**

Tabla 8. Presupuesto total estimado inicialmente para el proyecto

3. Conocimientos previos

En este capítulo explicaré de forma general las distintas tecnologías y herramientas usadas para desarrollar el proyecto. No se han usado muchas, ya que el código del sistema se ha hecho todo desde cero.

3.1. Código de *Armagetron Advanced*

Tal y como se ha mencionado en la descripción de los objetivos en el capítulo 2, el proyecto tenía que encontrar un juego de código abierto para así poder modificarlo e incluirle la opción de usar nuestro sistema. Después de mirar un poco los que había disponibles (que no son muchos, ya que la mayoría de juegos no son libres porque tienen como objetivo sacar beneficio), el que más me llamo la atención fue el *Armagetron Advanced* (Figura 4) [12].

Figura 5. Logo del juego que se usará para evaluar el sistema

Todo el código del juego está bajo una licencia GPL, por lo que es completamente libre. Se trata de un juego de acción basado en la película *Tron* (1982) en el que el jugador controla un vehículo que deja una estela. Si chocas contra la estela de uno de los jugadores contrarios o contra alguna de las paredes del recinto pierdes la partida (Figura 5). El objetivo del juego es ser el único superviviente. También se puede jugar por equipos.

El código del juego está en C++, que es uno de los requisitos que quería que tuviera el videojuego elegido, ya que como he dicho en el apartado de “Motivaciones”, quería usar este lenguaje para aumentar los conocimientos que ya tenía. Además, el código del sistema también es en C++, así que integrarlo ha sido sencillo.

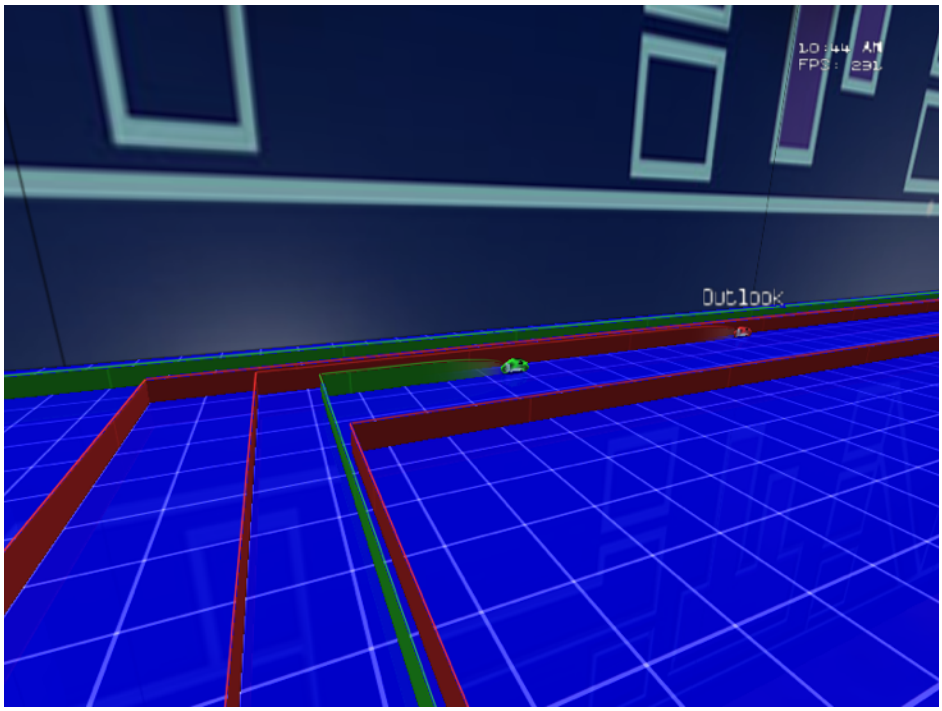

Figura 6. Ejemplo de partida de Armagetron Advanced

Tuve que estudiar el código del juego para poder encontrar el lugar donde integrar el sistema y comprobar si se podía reusar algo del código ya existente. El juego está estructurado en carpetas de la siguiente manera:

```
/src
        /engine
        /network
                /quickplay
        /render
        /tools
        /tron
        /ui
```

Como el proyecto trata sobre implementar un nuevo tipo de conexión online para el juego, es lógico que en donde más se ha centrado el trabajo sea en la carpeta de *network*. En esta carpeta está el código del juego que sirve para enviar mensajes a los otros jugadores o servidores, con información para llevar a cabo las partidas. También se encuentra el código para el servidor principal del juego que mantiene las partidas a través de Internet. Está todo estructurado en distintas clases para definir los distintos

objetos útiles (encargados de abrir los *sockets*, leer los mensajes que se reciben, enviarlos, entre otras cosas). Dentro de este directorio se ha creado la carpeta "*quickplay*", que es donde he integrado el sistema que se ha desarrollado.

En la carpeta *tools* se encuentran objetos que se usan a lo largo del código del juego, como por ejemplo su propia versión de "string" o una lista enlazada. Sólo he necesitado saber de alguno de ellos, ya que era necesario para poder imprimir los mensajes del estado de la conexión al buscar partida. También para crear los elementos que aparecen en los menús, necesarios para poder usar el sistema y buscar partidas con otros jugadores.

La carpeta más central es la de *tron*. Aquí es donde se encuentra el punto de entrada de ejecución del juego (función *main*), así como las clases para definir la configuración del juego, el jugador o la partida. Son las funciones de la clase *game.h* las que he tenido que estudiar para poder iniciar la partida una vez se reciben las órdenes del servidor del sistema.

Finalmente, la carpeta *ui* es la que contiene el código necesario para crear los menús del juego. Ha sido necesario saber cómo funcionaban para poder mostrar las distintas opciones añadidas al juego.

3.2. C++

C++ es un lenguaje de Programación Orientado a Objetos [13]. Este tipo de programación considera que cada elemento del programa desarrollado es un objeto, con sus propiedades y métodos. Pero C++ es mucho más que eso. A continuación presento algunas de sus principales características:

- Es un lenguaje estandarizado desde 1998 por la ISO (*International Organization for Standardization*).
- Es compilado. Esto significa que otro programa lo traduce directamente a código máquina nativo, lo que permite que sea uno de los lenguajes más rápidos del mundo si se optimiza.

- Es fuertemente tipado. Esto significa que requiere que el programador sepa lo que hace, obligándole a definir el tipo de cada variable, por ejemplo. El resultado es un gran control del programa.
- Es portable. Al ser uno de los lenguajes más usados y abierto, C++ tiene un amplio abanico de compiladores capaces de ejecutarse en diferentes plataformas que le dan soporte.
- Tiene una gran librería disponible. Esta librería incluye funciones y objetos de todo tipo que ahorran trabajo al programador.

Tanto el código del juego como del sistema están hechos con este lenguaje de programación, así que he tenido que dedicar una buena cantidad de tiempo a repasar conceptos y a aprender como usar el lenguaje, sobre todo aquello relacionado con la POO, que no he llegado a usar mucho durante el grado.

3.3. Autotools

Autotools [14] es el término que se usa para un grupo de herramientas desarrolladas en los 90 por el proyecto GNU: *Autoconf*, *Automake* y *Libtool*. Las tres fueron desarrolladas separadamente para abordar el problema de hacer la configuración del software más manejable dividiéndolo. Aunque fueran desarrolladas separadamente están diseñadas para ser usadas como un único sistema, de ahí está agrupación.

Estás herramientas son paquetes que permiten que el software sea más portable y hacen que sea más sencillo compilarlo (ya sea en un sistema propio o de alguna otra persona). Las he necesitado para desarrollar el proyecto porque el juego las usa, ya que es portable a cualquier plataforma (Linux, Mac o Windows) gracias a Autotools. También he podido aprovechar los ficheros de configuración ya creados en el propio juego para poder añadir simples instrucciones que permitiesen compilar mi sistema sin necesidad de crear otros ficheros nuevos de configuración, ahorrándome así un trabajo extra. A continuación daré una breve explicación de cuál es el propósito de cada una de estas utilidades.

Autoconf sirve para hacer que los paquetes sean más portables haciendo una serie de tests para descubrir características del sistema antes de compilar el programa. El código del programa puede adaptarse a estas diferencias.

Automake es una herramienta que sirve para generar *Makefiles*. Un *Makefile* es un fichero que describe qué se tiene que compilar y en qué orden para poder construir correctamente el programa a partir del código escrito. *Automake* simplifica mucho el proceso de describir la organización de un paquete de software y realiza funciones adicionales como el rastreo de dependencias entre ficheros de código fuente.

Libtool es una interfaz mediante línea de comandos hacia el compilador y el enlazador (*linker*) que hace que sea fácil generar librerías estáticas y compartidas, independientemente de la plataforma en la que se ejecute.

A parte de estas 3 herramientas también hay otras muy útiles que se usan en conjunto con ellas. Una que usa el proyecto es *Autoheader*. Este paquete utiliza el mismo fichero de configuración que *Autoconf* para detectar posibles definiciones de C++ y genera un *header* con ellas. Por ejemplo, podemos usar un `#ifdef WINDOWS` en nuestro código y cuando *Autoheader* se ejecute definirá la variable en el fichero de salida (normalmente config.h.in) sólo si estamos en un sistema WINDOWS. Con un `#else` más adelante nos encargamos de considerar los casos restantes.

Gracias a todos estos paquetes el código del juego es portable, por lo que las he usado también para el desarrollo del sistema de este proyecto. Lo único que se necesita para usarlas son 2 ficheros de configuración: configure.in y Makefile.am. En ellos se especifica, siguiendo las instrucciones de *Autotools,* qué tests se quieren ejecutar, qué definiciones hay que realizar y qué ficheros se quieren compilar (de las dependencias entre ellos se encarga *automake*).

3.4. Git

Tal y como se ha mencionado en el apartado de metodología de trabajo, en este proyecto se ha usado Git para disponer de un sistema de control de versiones distribuido. Git es un software diseñado para mantener distintas versiones de aplicaciones con un gran número de archivos de código fuente [15]. Esta herramienta

permite tener un diario de todos los cambios que se han ido realizando a lo largo del desarrollo y volver a cualquiera de estos puntos.

Git también permite crear distintas ramas de trabajo para probar distintos métodos de solucionar problemas sin comprometer el código base para que este pueda ser usado por otros. Además, se ha creado un repositorio en GitHub para poder tener el código del proyecto en Internet y de esta manera poder acceder fácilmente desde cualquier lugar y ordenador.

Es una herramienta sencilla de utilizar y muy usada en el mundo laboral, por lo que es bueno conocerla bien.

4. Especificación

Todo proyecto de software dispone de una especificación, que es necesaria para poder empezar a escribir el código. La especificación consiste en aquellos documentos que proporcionan la información de las necesidades que ha de satisfacer el proyecto y de sus características esenciales. A continuación se detallan la especificación del sistema desarrollado y la del protocolo de conexión que este usa.

4.1. Descripción del sistema

El sistema desarrollado es un *matchmaker*, que significa “generador de partidas”. Su objetivo es mantener una lista de todos los clientes (en el caso de un juego serán los jugadores) que quieran participar en una partida y juntarlos en el número adecuado para que las partidas puedan tener lugar. Además, este sistema no mantiene las partidas en el servidor como en muchos otros casos actualmente, sino que informa a los clientes de quienes son sus compañeros para que puedan crear la partida y comunicarse sin necesidad del servidor. De esta descripción del sistema se puede deducir que este consta de dos partes principales: el código del cliente y del servidor.

Cliente

Para empezar, el código del cliente debe enviar un mensaje al servidor con información sobre él mismo. Esta información consiste en datos sobre la CPU; incluye el número de procesadores y la velocidad de estos en GHz. Esta información será usada por el servidor a la hora de determinar quién creará la partida, ya que necesitará tener un mejor ordenador para procesar la información de los jugadores.

Otro factor a tener en cuenta es la latencia o ping entre los jugadores. Esta parte se ha dejado como futura mejora por desarrollar y dado que no hay ninguna manera de determinar en que localización se encuentran los jugadores, para disminuir el tiempo que tardan en llegar los mensajes se recomienda poner un servidor por región. De esta

manera los jugadores que el servidor junte estarán relativamente cerca, asegurando así que el ping entre ellos no sea muy elevado.

Después de enviar la información sobre su CPU, el cliente espera. Mientras espera se irán imprimiendo mensajes por la pantalla para que el jugador no se piense que el juego se ha quedado congelado. Cuando el servidor haya encontrado rivales y compañeros para el jugador le enviará un mensaje con su función a llevar a cabo. Estas pueden ser:

1) El jugador tiene que crear la partida. En este caso el jugador hará de servidor (a la vez que jugador) para el resto de participantes de la partida. Esto implica que una vez reciba el mensaje tendrá que crear una partida usando el código para hacer de servidor del cliente. Una vez la partida esté creada y lista para recibir al resto de jugadores enviará un mensaje al servidor indicando este estado y se desconectará de él. A partir de entonces tendrá que esperar hasta que los jugadores restantes se hayan conectado para poder empezar la partida.

2) El jugador se une a una partida creada por otro jugador. En este caso el mensaje que recibirá el cliente que está a la espera llevará como información la dirección a la que se debe conectar. Una vez recibida la información dará el acuse de recibo (ACK) al servidor y se desconectará de él. Finalmente, se conectará a la partida creada mediante la función correspondiente.

Servidor

El funcionamiento del servidor es un poco más complejo, ya que debe realizar más tareas que los clientes. El servidor estará escuchando en un puerto (3490 por defecto) y esperará a recibir jugadores. Cuando un jugador abre una nueva conexión con el servidor es puesto en una lista de jugadores con todos aquellos que están esperando para empezar una partida.

Después de añadir el jugador a la lista de espera el servidor no hace nada más, ya que tiene que esperar a que el jugador le envíe la información sobre su CPU. Una vez le ha llegado el mensaje con dicha información, el servidor considera al jugador como apto para unirse a una partida y lo tendrá en cuenta cuando tenga que juntar a diversos jugadores.

Cuando haya suficientes jugadores aptos para formar una nueva partida el servidor los añadirá a una lista de partidas con un ID único representando el juego. Comprobará cuál de ellos tiene el mejor procesador para que sea el encargado de crear la partida (en caso de empate lo seleccionará aleatoriamente) y le enviará a ese jugador un mensaje indicándole que es el que debe crear la partida mientras mantiene al resto en espera.

El servidor está permanentemente escuchando por si recibe más conexiones o algún mensaje de otro jugador. Cuando le llega el mensaje de que la partida está lista, comprueba el ID de la partida y avisa al resto de jugadores que tengan el mismo ID asignado. En el mensaje incluye la dirección del propietario de la partida. Una vez los jugadores hayan recibido el mensaje y se hayan conectado a la partida se desconectan del servidor. El creador de la partida se desconecta al avisar al servidor de que está preparado para recibir al resto de jugadores.

Cuando un jugador se desconecta del servidor es borrado de la lista de jugadores, y si estaba en la lista de partidas también se le borra de esa lista. Cuando todos los jugadores con un mismo ID de partida han sido eliminados quiere decir que la partida está en marcha. Si queda alguno por eliminar quiere decir que algún mensaje se ha perdido por el camino, en cuyo caso habrá que reenviarlo.

4.2. Protocolo de conexión

Para asegurar que todos los mensajes son recibidos correctamente, tanto por parte del servidor como del cliente se deberá crear un sistema de *acknowledgement* (acuse de recibo). Cada mensaje será respondido conforme se ha recibido correctamente. Los mensajes enviados no se borrarán, sino que se almacenarán en un buffer por si acaso hay algún problema en la red y tiene que ser reenviado.

Además, para facilitar el reenvío de posibles fragmentos de paquetes perdidos y asegurar que lleguen en orden a su destino se usa el protocolo TCP en todas las conexiones con el servidor del sistema. Más adelante, al conectarse a la partida cada juego es libre de usar otro tipo de conexión diferente, ya que muchos usan UDP. Esto se debe a que el servidor de la partida no tiene que esperar a reenviar un mensaje perdido

si ya se ha recibido el siguiente, cosa que haría que la partida se congelase y provocaría una mala experiencia de juego a los usuarios.

Cada mensaje tendrá un tipo y una longitud de mensaje como cabecera. Así se puede saber cuando un mensaje ha sido recibido completamente, en el caso de que este se haya fragmentado. Teniendo en cuenta cuáles son las distintas tareas a realizar, habrá 6 tipos de mensaje distintos.

1) *Ack*. Este es el tipo de mensaje que se envían, tanto el cliente como el servidor, para indicar que han recibido un mensaje correctamente.
2) *Player Information*. Este es el mensaje que envían los jugadores al conectarse al servidor para informarle de los datos necesarios sobre su CPU.
3) *Match Ready*. Mensaje que envía el cliente encargado de crear la partida al servidor cuando ya está lista para recibir al resto de jugadores.
4) *Resend*. Este es el mensaje que envían los clientes al servidor cuando les falta información por recibir o cuando la han recibido con errores. El servidor nunca pide que se le reenvíe información; son los clientes quienes, con un cronómetro para dejar pasar un tiempo establecido, reenvían la información al servidor si este no les ha contestado con un ACK.
5) *Hosting Order*. El servidor envía este tipo de mensaje a un cliente para indicarle que es él quien debe crear la partida.
6) *Connection Information*. Este mensaje es enviado por el servidor a los clientes de una partida que no son su creador. Contiene la dirección a la que se deben conectar.

Una vez especificados los distintos tipos de mensajes que se enviarán, sólo queda por definir el orden en que se hará. Para empezar, el servidor estará escuchando en la dirección y puerto introducidos en el código del cliente. Los clientes que quieran usar el servicio para buscar una partida a través de Internet abrirán una nueva conexión al servidor. Luego de abrir la conexión enviarán un mensaje de tipo *Player Information*. Si el servidor recibe el mensaje correctamente, entonces responderá con un *Ack*. Si no lo ha recibido correctamente por el motivo que sea, no hará nada. El cliente, al enviar el mensaje inicia una cuenta atrás; si cuando la cuenta atrás llega a 0 aún no ha recibido el mensaje de *Ack* del servidor, vuelve a enviar el mismo mensaje. Este proceso se repite un cierto número de veces y si no se consigue enviar correctamente el mensaje se cierra la conexión y se muestra un mensaje de error por pantalla.

El proceso explicado anteriormente lo realizan todos los clientes al iniciar el sistema de este proyecto. Si el mensaje es recibido correctamente, el jugador estará disponible para ser elegido por el servidor para unirse o crear una partida. Permanecerá esperando durante un tiempo mientras está en cola a la espera de otros jugadores. Si pasa demasiado tiempo a la espera, el servidor borrará el jugador de la lista de espera. Esto hará que se imprima un mensaje de error en la pantalla del jugador y si este lo desea podrá volver a ponerse en cola.

Una vez hay suficientes jugadores para iniciar una partida el servidor los une asignándoles su ID de juego único y elige cuál va a ser el encargado de hacer la función de servidor del juego. A ese jugador le envía el mensaje de *Send Hosting Order*, mientras el resto de jugadores sigue a la espera. Si este mensaje se pierde, el servidor tampoco lo reenvía. Son los propios clientes los que, cada "X" tiempo, van enviando mensajes de tipo *Resend* al servidor, pidiéndole que se les reenvíe la información. Si el servidor no tiene nada que enviar ignora el mensaje. En caso de que el mensaje se haya perdido, entonces el servidor lo tendrá guardado en la cola de "pendientes de *Ack*", en cuyo caso lo reenviará.

La decisión de que el servidor no disponga de la opción de reenviar los mensajes se ha tomado para que no tenga que realizar trabajo extra. De esta forma los clientes deberán esperar algunos segundos más en caso de que algún mensaje se pierda, pero esto se compensa con el hecho de que el servidor podrá atender a más clientes a la vez.

Cuando el jugador recibe el mensaje de *Send Hosting Order* correctamente y ha iniciado la partida, envía un mensaje *Match Ready* de vuelta al servidor. Cuando recibe el *Ack* del servidor se desconecta y espera a que el resto de jugadores se unan a la partida para empezar.

El servidor recibe el mensaje de *Match Ready*, y responde con un *Ack*. Entonces comprueba el número de partida del jugador que ha enviado el mensaje y crea un mensaje de tipo *Send Connection Information* con la dirección del jugador. Este mensaje lo envía al resto de jugadores con el mismo número de partida. A medida que van llegando sus *Ack*'s los va borrando de su memoria, ya que quiere decir que ya se han unido a la partida.

Para ayudar a comprender este proceso de envío de mensajes, he creado una ilustración en la que se pueden ver los distintos pasos que se van sucediendo, ordenados en orden cronológico mediante números (Figura 6). En esta imagen se considera que el número de jugadores necesario para realizar una partida son 2 (1 vs 1) y que no sucede ningún error de red.

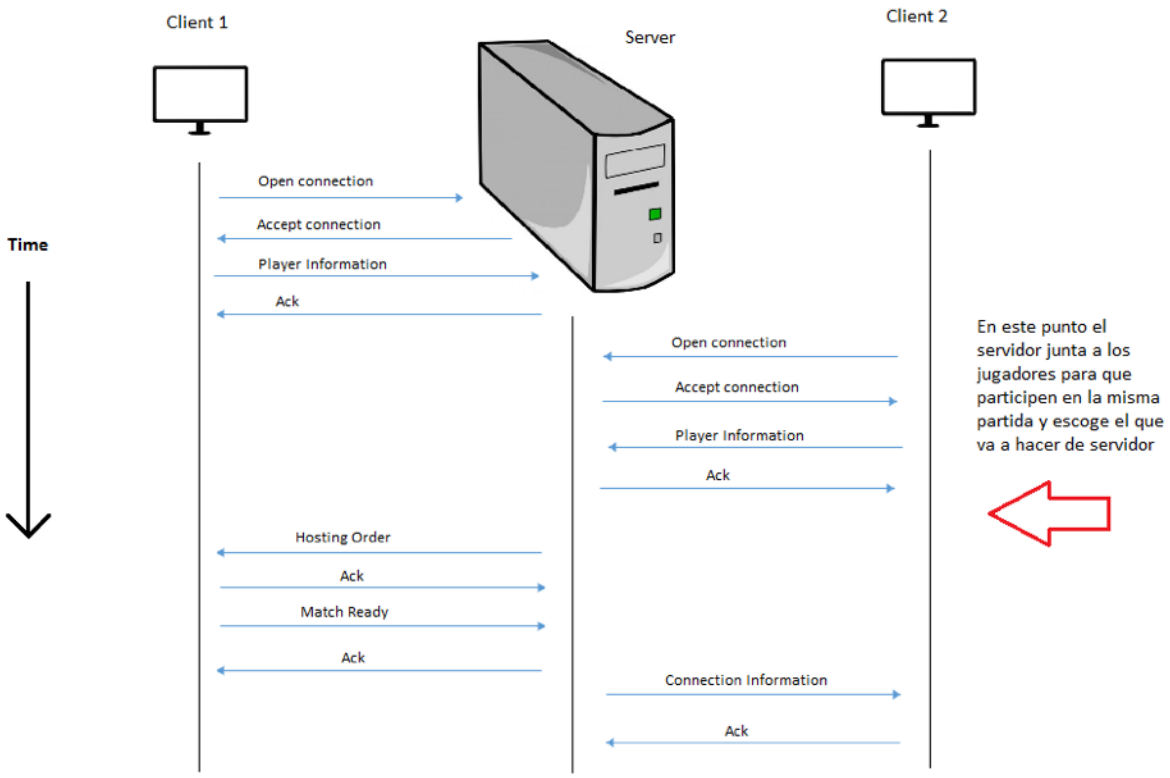

Figura 7. Esquema del envío de mensajes entre servidor y clientes

5. Implementación

En este capítulo se explicarán brevemente algunos detalles de la implementación realizada, pero sin centrarse en el código escrito. Para ver el código del sistema se puede consultar el repositorio público de GitHub del proyecto (https://github.com/alvaromateo/armagetronad), que se encuentra bajo una licencia GPL, o la sección de apéndices, donde se encuentran algunas de las clases utilizadas.

Los únicos programas que se han usado para esta parte son *Sublime Text 2* (Figura 7) para la edición del código fuente y la consola iTerm para compilar y ejecutar el programa. El iTerm lleva un plugin llamado *Oh My Zsh* (creado por Robby Russell) cuyo único propósito es mejorar la apariencia visual de la terminal (Figura 8).

Antes de empezar a hablar de cómo está implementado el sistema, tengo que comentar que todos los nombres de clases pertenecientes al código de este software se han precedido con una “q” (de *quickplay*) para diferenciarlos de otros posibles nombres iguales en algún videojuego que quiera usar este proyecto.

Lo primero que detallaré sobre la implementación será todo lo relacionado con los mensajes, ya que es común tanto para el cliente como el servidor. Los dos usan las mismas clases y métodos para gestionar los mensajes recibidos y enviados, aunque luego añadan alguna funcionalidad específica más, si se necesita.

qMessage

Para definir los mensajes se ha utilizado una clase genérica que los engloba a todos: *qMessage* (Apéndice A.1.). Los mensajes constan de un buffer en el que se escribe la información que debe transmitir el mensaje, si es requerido por el tipo de mensaje usado. A parte del buffer con información, contienen la longitud del mensaje, un byte para indicar el tipo de mensaje y un short que indica la siguiente posición del buffer a leer o escribir. Los 3 primeros bytes de un mensaje siempre serán el tipo (1 byte) y la longitud del mensaje (2 bytes).

Esta clase contiene métodos para consultar cualquiera de sus variables y métodos para cambiar la longitud del mensaje (*messLen*) y la posición de lectura del buffer (*currentLen*). El método que permite obtener el buffer devuelve un puntero no constante, por lo que es posible de modificar el buffer a partir de ahí. El tipo de mensaje no puede ser cambiado. Una vez se crea un mensaje de un tipo con el constructor de la clase siempre seguirá perteneciendo a ese tipo; lo único que puede cambiar es su contenido.

Además, la clase *qMessage* ofrece métodos para leer y escribir números en el buffer y para añadir nueva información a un mensaje que se ha recibido a medias. También tiene un método para comprobar si el mensaje se ha recibido completamente (comprobando la longitud que se tiene con la indicada en la cabecera del mensaje). Un método también muy útil es *acknowledgeMessage*, que se encarga de enviar automáticamente al dueño del mensaje el acuse de recibo.

Finalmente, la última parte de la clase son los métodos virtuales *handleMessage*, *send* y *prepareToSend*. El primero se encarga de inicializar las variables de un mensaje sin inicializar a partir del buffer que *recv* (función del sistema) [16] se encarga de llenar. El segundo se encarga de enviar a través de la red el mensaje, que debe estar correctamente escrito, para lo que se usa *prepareToSend* antes. Cada tipo de mensaje tiene una subclase que se encarga de inicializar esta con el tipo correcto.

Al recibir un mensaje se mira el primer byte y se crea el tipo de mensaje adecuado para que luego el objeto pueda ser inicializado por *handleMessage*, a medida que se va recibiendo la información necesaria. De esta forma se consigue tener en cuenta los casos en los que el paquete se pueda fragmentar (aunque al no ser paquetes demasiado grandes no suele ocurrir).

En cuanto a la función *send* de la clase, esta también se encarga con un bucle de que toda la información del buffer sea transmitida, ya sea en un solo paquete o en múltiples.

qMessageStorage

qMessageStorage es la clase encargada de guardar los mensajes que se van recibiendo, los que están listos para ser enviados y los que están pendientes de ser confirmados como recibidos. Con este propósito, esta clase contiene 3 mapas

(*std::map*) que contienen los mensajes en cada uno de los estados mencionados. Se pueden ver las variables y métodos que contiene esta clase en el Apéndice A.2.

Estos *maps* tienen como llave el entero de la conexión (*socket)* al que va destinado el mensaje o del que ha llegado. Como valor guardan un puntero al mensaje en cuestión. Consecuentemente, este diseño sólo puede almacenar 3 mensajes por conexión, uno en cada mapa. Esto no es ningún problema, ya que todos los mensajes son enviados de 1 en 1 y hay que esperar al *Ack* antes de poder enviar otro, así que no se pueden acumular 2 mensajes seguidos para una misma conexión.

Hay que puntualizar que en el caso de los clientes el mapa tendrá una entrada nada más, correspondiente a la conexión abierta para comunicarse con el servidor. El servidor en cambio, tendrá muchas más, dependiendo de cuantos clientes a la vez estén solicitando su servicio.

Tiene funciones para poder acceder a los 3 mapas y una que devuelve un iterador a un elemento, dado el entero de conexión y el mapa en el que se quiere buscar. También tiene funciones para borrar, añadir y mover mensajes de los mapas. Además, también posee la función *handleData* que es la encargada de crear los distintos tipos de mensajes y añadirles la información que se va recibiendo. Esta es la función que llaman tanto el servidor como el cliente cuando han leído información y la tienen almacenada en un buffer temporal. Con el método *createMessage* se pueden crear mensajes de distintos tipos según el byte que se le proporcione.

Finalmente, tiene dos métodos que permiten el envío de mensajes. *sendMessages* se encarga de enviar todos los mensajes que se encuentren pendientes de enviar; itera por todo el mapa y va llamando al método *send* de cada mensaje. *resendUnacked* mueve los mensajes pendientes de *Ack* de vuelta a pendientes de enviar para volver a transmitirlos.

Estas dos clases son las que comparten tanto el código del servidor como el del cliente. Los dos extienden de *qMessageStorage*, ya que la necesitan para tratar todas las conexiones y mensajes que reciben. Algunos de los métodos son virtuales para que puedan ser redefinidos en el caso de que se necesite un comportamiento diferente dependiendo de si se está ejecutando el cliente o el servidor. A continuación se encuentran las descripciones para la implementación del código del servidor y el del cliente.

5.1. Servidor

El código del servidor está dividido en 2 partes. Primero está la clase base *qServer* que contiene todo lo necesario para iniciar un servidor que acepte conexiones entrantes y sea capaz de mantener la comunicación con ellas. La clase *qServerInstance* es la que se usa para iniciar el servidor de nuestro servicio. Extiende de la clase *qServer,* para obtener las variables y métodos que le permitan conectarse con los clientes, y de la clase *qMessageStorage* para poder gestionar los mensajes recibidos. Se puede consultar la definición de estas dos clases en el Apéndice A.3.

La clase *qServer* es sencilla. Contiene un entero para indicar el *socket* en el cuál el servidor escuchará para atender a conexiones entrantes. También dispone de un entero para guardar el *file descriptor* (descriptor de fichero) más alto abierto y un grupo de *file descriptors* que contiene todos los *sockets* que están abiertos con conexiones a distintos clientes. Tiene 3 métodos para consultar cada una de estas 3 variables y un método puramente virtual (tiene que ser definido en la clase que extiende de esta) para obtener la información de la red. Es en el constructor de esta clase donde se inicia el la escucha del servidor.

Se han separado estas funcionalidades y las de gestión de mensajes del resto de tareas que son realizadas por el servidor porque este diseño permite crear fácilmente un nuevo servidor que realice distintas tareas, en caso de que sea necesario. Así el software del sistema es fácilmente modificable, si se quieren añadir nuevos tipos de servidores, sin tener que cambiar completamente el diseño del sistema ya creado.

En la clase *qServerInstance* es dónde se desarrolla todo el proceso de recepción de la información, envío de mensajes y creación de las partidas. Dispone de un entero estático (*static*), que sirve para asignar el número de ID único a cada partida creada, y dos mapas (un *std:map* y un *std:multimap* para ser más exactos), una para los jugadores conectados al servidor en cada momento y la otra para las partidas que están en proceso de ser creadas.

Como casi todas las clases, dispone de métodos para obtener sus variables, que son privadas debido a la encapsulación del diseño. También tiene métodos para obtener, añadir y borrar jugadores de cualquiera de los dos mapas.

Es en esta clase donde se encuentra definido el método *getData*, que es el encargado de leer la información de la red para luego llamar a los métodos de la clase *qMessageStorage* que permiten guardar el mensaje. El método *processMessages* es el encargado de realizar las acciones que se deban realizar una vez los mensajes que se han recibido están completos. Finalmente, *prepareMatch* es el método que comprueba si hay suficientes jugadores disponibles para crear una partida nueva y si así es, procede a decidir quién será el que actuará como servidor y a enviar los mensajes correspondientes. Las partidas creadas se añaden al *multimap*, que tiene como llave el ID de la misma y como elementos punteros a los distintos jugadores que participan en ella.

Una vez definidas las clases que intervienen en el el servidor, voy a explicar ahora como funciona este. Como todo servidor, la base es un bucle infinito que se encarga de escuchar a las conexiones entrantes. Dentro de este bucle el servidor debe realizar 4 tareas:

```
processMessages();
sendMessages();
prepareMatch();
getData();
```

El primer método en ejecutarse es *processMessages*. Este método recorre los mensajes recibidos y comprueba si están completos. Si sólo se ha recibido una parte del mensaje y está incompleto entonces no hace nada con él; ya se encargará de procesarlo cuando se haya recibido la parte restante. Si por el contrario está completo, entonces inicializa las variables del mensaje con la información disponible en el buffer y realiza las acciones necesarias según su tipo. Estas pueden ser: añadir información sobre la CPU de uno de los jugadores que tienen una conexión abierta (de esta manera pasan a estar disponibles para ser elegidos como participantes de una partida) o enviar un mensaje con la dirección de la partida creada por un cliente al resto de jugadores de esa partida. En el caso de que el mensaje recibido sea de *Ack*, al procesarlo se borra del mapa de mensajes pendientes de acuse de recibo el correspondiente *qMessage*. Después de procesar un mensaje recibido este es borrado del mapa de mensajes recibidos.

Después se ejecuta *sendMessages*. Este método es similar al anterior, pero recorre los mensajes listos para enviar en lugar de los recibidos. Cada mensaje que se encuentra es enviado a través de la red.

A continuación se preparan las partidas con el método *prepareMatch*. Este comprueba cuantos jugadores con información sobre su CPU hay disponibles y los junta en grupos de tantos como se haya especificado que deban haber en una partida. Para cada grupo de jugadores comprueba cuál es el que tiene un mejor procesador y lo escoge para ser el *master* de la partida. Para ello crea un mensaje del tipo *Hosting Order* y lo añade al mapa de mensajes listos para enviar con la llave del entero perteneciente a la conexión del jugador elegido. Al resto de jugadores no les envía nada; cuando llegue el mensaje de *Match Ready* del cliente escogido entonces les enviará la dirección al resto de jugadores.

Finalmente, *getData* se encarga de recoger la información mediante la llamada a la función *select* [17]. Una vez hay información disponible para ser leída, esta es utilizada para crear un mensaje y añadirlo al mapa de mensajes recibidos. Esta es la última función en ser realizada ya que *select* bloquea el proceso hasta que haya información para ser leída o haya una nueva conexión/desconexión.

La eliminación de jugadores y de partidas de la memoria del servidor se realiza cuando los clientes deciden terminar la conexión con el servidor debido a que ya están listos para empezar la partida. Es en el último método en el que, cuando se detecta que se ha cerrado una conexión, se eliminan todas las referencias al cliente que se ha desconectado.

5.2. Cliente

El diseño es igual que el del servidor. Existe una clase base *qConnection* con todo lo necesario para conectarse al servidor y una clase que la extiende llamada *qPlayer* con todas las variables y métodos que realizarán las tareas que sean precisas para leer y escribir la información que será enviada al servidor. La clase *qPlayer* también extiende de *qMessageStorage*, para así aprovechar lo que ya se había realizado para el servidor. Aunque en el caso del cliente los mapas tendrán como mucho un mensaje, ya que sólo habrá una conexión abierta (la que le permite comunicarse con el servidor). La definición de estas dos clases se puede ver en el Apéndice A.4.

En la clase *qConnection* sólo se necesita un entero para guardar el descriptor de fichero de la conexión con el servidor y una estructura capaz de almacenar la dirección a la que se tendrá que conectar el cliente para jugar la partida (*sockaddr_storage*). Esta estructura es capaz de guardar tanto una dirección IPv6 como una IPv4. Los métodos que se encargan de transmitir y recibir la información sobre la dirección también están preparados para los dos tipos.

Además, la clase dispone de métodos para obtener sus variables (privadas), un método para comprobar si hay una conexión abierta y otro para poder cerrar la conexión en el caso de que el cliente haya terminado la comunicación con el servidor y ya sepa la dirección a la que tiene que conectarse para jugar la partida.

En cuanto a la clase *qPlayer*, esta contiene un objeto con información sobre la CPU del jugador (del cual se hablará en el siguiente punto de este capítulo), un entero con el número de identificación de la partida que le haya asignado el servidor del sistema (este número será 0 si el jugador aún no está asignado a ninguna partida) y un *boolean* indicando si el jugador ha de actuar como creador de la partida o no.

Esta clase tiene una peculiaridad importante en sus 2 constructores. Uno no acepta parámetros (constructor por defecto) y es el que llaman los clientes para iniciar la conexión con el servidor, a parte del resto de variables del objeto. El otro tiene como parámetros el entero para almacenar el descriptor de fichero asignado a la conexión y una dirección. Este último es el constructor que llama el servidor cuando quiere guardar un jugador en su memoria, pasando como parámetros el *socket* que tiene asignado y la dirección del cliente. Lo único que hace este constructor es inicializar las variables con los valores proporcionados.

Con este diseño se puede aprovechar el código usado en el servidor para almacenar jugadores para el código del cliente. Sólo hay que añadirle a la clase *qPlayer* los métodos necesarios para que pueda leer y enviar mensajes. Los objetos de esta clase definidos en el servidor no usan estos métodos ni tienen ningún mensaje guardado en su *qMessageStorage*.

A parte de los constructores, la clase también dispone de métodos para obtener y cambiar sus variables privadas que pueden ser usados tanto por el cliente como por el servidor. Tiene su propia definición de los métodos *processMessages* y *getData*, ya que realizan diferentes tareas que el servidor, y un método para comprobar que no haya

ningún mensaje pendiente de *Ack*. También dispone de un método para enviar información sobre la CPU del jugador al servidor y otro para enviarle un mensaje indicando que la partida creada está lista. Estos métodos son usados sólo en el código del cliente. Finalmente, dispone de un método más que es el que usa el servidor para comprobar si la CPU de un jugador es mejor que la de otro.

Los pasos que se siguen en el cliente son un poco más complejos que un bucle con 4 métodos distintos. Hay que realizar diferentes funciones y hay más posibilidades a tener en cuenta. Al pulsar el botón que se desee en la interfaz del juego se deberá llamar a la función *BrowseOnlineGame*. Esta función se debe integrar en una zona del código desde la que se pueda crear una partida y realiza los siguientes pasos:

1. Se inicia una variable *qPlayer* con el constructor por defecto para abrir la conexión con el servidor.
2. Se usa el método *sendMyInfoToServer* para recopilar datos sobre la CPU y enviarlos al servidor.
3. Se inicia un bucle en el que se espera a recibir algún mensaje del servidor. Este funciona de forma parecida al del servidor. Primero se llama a *processMessages* y *sendMessages*. A continuación se llama a *getData*, pero este tiene una diferencia comparado con el del servidor. El *select* [17] que en el se usa no se bloquea indefinidamente hasta que llegue un mensaje, ya que si tarda el jugador se puede pensar que se le ha congelado el juego. En lugar de eso, retorna al cabo de 10 segundos si no se ha recibido nada. Si se superan 15 veces sin recibir información del servidor entonces se aborta la búsqueda de la partida. Mientras no se hayan superado, si el cliente dispone de algún mensaje pendiente de *Ack*, lo reenviará.
4. Una vez el servidor responda se pueden dar 2 casos:
   4.1. El cliente recibe un mensaje de tipo *Hosting Order* y debe crear la partida. En el caso de que el cliente reciba este mensaje, el método *processMessages* de *qPlayer* dará a la variable *master* valor TRUE (verdadero). Esto hará que se salga del bucle.
   4.2. El cliente recibe un mensaje de tipo *Connection Information*. En este caso en el método de *processMessages* se dará valor a la variable que contiene la dirección a la que debe conectarse el jugador para participar en la partida. Esto también provocará que se salga del bucle.
5. Si se ha dado el caso de que el mensaje recibido era de *Hosting Order*, entonces se deben realizar dos tareas. Iniciar la partida y avisar al servidor del momento en el

que la partida está preparada para aceptar las conexiones del resto de jugadores. Para ello he decidido crear un *thread* (hilo) que se encargue de enviar el mensaje cuando la partida esté lista, mientras el *thread* principal se encarga de toda la inicialización usando el código del juego. Cuando la partida está lista para recibir conexiones del resto de jugadores se indica mediante el valor de un *boolean* global. El *thread* encargado de enviar el mensaje al servidor estará esperando a que esta variable sea verdadera y cuando eso suceda continuará su ejecución para enviar el mensaje y cuando recibe el *Ack* del servidor finaliza su ejecución. No hace falta volver a unir el *thread* al principal, así que al acabar, este se destruye sin necesidad de sincronizar ambos. Antes de destruirse, el *thread* cierra la conexión al servidor después de haber recibido el *Ack*. Mientras no lo ha recibido continua enviando el mensaje de *Match Ready* de la misma forma que si de otro mensaje se tratase.

6. Si se ha recibido el mensaje con información de la dirección a la que conectarse el procedimiento es más sencillo. Se contesta con un *Ack* al servidor y se cierra la conexión. Para acabar, se usa la función del juego adecuada para conectarse a una partida dada una dirección.
7. Cuando todos los jugadores se han conectado a la partida creada el código del juego se encarga de que esta de comienzo.

La eliminación de las variables usadas (*qPlayer* y alguna variable temporal más) se realiza cuando se finaliza la partida y la función acaba su ejecución. Los destructores de cada variable se encargan de borrar todo rastro de objetos creados en el *heap*.

5.3. Información sobre CPU

Una de las partes básicas del sistema consiste en conseguir que el jugador con mejor CPU sea el que crea la partida, ya que no tendrá tantos problemas para hacer de servidor y cliente a la vez. A lo largo de la descripción de la especificación y la implementación se ha mencionado que se recopila información sobre la CPU de los jugadores y se envía al servidor para que este decida quién debe crear una partida. A continuación se explican brevemente los métodos que se usan para leer esta información. En el Apéndice A.5. se puede encontrar la definición de la clase *PlayerInfo*, que es la encargada de almacenar toda la información relevante a la CPU.

Como se puede ver en la definición del Apéndice, la clase dispone de 3 *unsigned char* (1 byte cada uno) como variables. Los números a almacenar no son elevados (no

pasarán de 255 en ningún caso) y así se ahorran bytes que enviar a través de la red. La información que almacenan son: el número de CPU's o *cores* del ordenador, la parte entera de su velocidad en GHz y la parte fraccional (dos dígitos como mucho).

La clase dispone de 2 constructores. El primero, que no acepta parámetros, inicializa todas las variables a 0. El segundo las inicializa con el valor proporcionado y es usado en el servidor, ya que los parámetros son recibidos por medio del mensaje que ha enviado el cliente. Para recopilar la información de la CPU se debe llamar explícitamente al método *setCpuInfo*. Además de este método, la clase dispone de *getters* (métodos para obtener las variables), *setters* (métodos para cambiar la información de las variables) y un método para comprobar si la información está inicializada o no.

La parte interesante de esta clase, y motivo por el cual tiene dedicado un apartado en esta memoria, es que como los juegos pueden funcionar en diferentes sistemas operativos, debe adaptarse al sistema operativo en el que esté. Mediante el paquete de *Autoheader* usado en el juego seleccionado se crea un fichero con definiciones; estas incluyen una que determina el sistema operativo en el que se va a ejecutar (`WINDOWS`, `MACOS` o `LINUX`). Si no se usa *Autoheader*, se deberá usar algún otro tipo de software que incluya estas definiciones, o incluirlas a mano en el caso de que el juego sólo vaya a estar disponible para un tipo determinado de plataforma.

El método *setCpuInfo* utiliza instrucciones del preprocesador (`#ifdef`) junto con las definiciones mencionadas para comprobar el sistema operativo en el que se va a ejecutar el juego y así compilar el trozo de código necesario para recopilar la información de la CPU (ya que esto se logra de maneras distintas en cada sistema operativo). A continuación se explica de manera resumida cómo se lleva a cabo la obtención de la información en cada uno de los sistemas operativos:

1) `WINDOWS` [18]. Cuando el juego se ejecuta en Windows es sencillo conseguir la información que se busca. Utilizando la función *GetSystemInfo* se recibe una estructura de la que se puede extraer el número de CPUs en el campo *dwNumberOfProcessors*. Para conocer la velocidad de la CPU hay que usar la función *QueryPerformanceFrequency*, que recibe como parámetro un puntero a un *long* (entero muy grande) y al acabar contiene el valor buscado, en Hz.

2) `MACOS` [19]. En MacOS también hay una llamada a sistema que devuelve la información buscada sobre la CPU: *sysctlbyname*. Esta función recibe como primer

parámetro el nombre de la información que se necesita. En el caso del número de CPUs será “hw.physicalcpu” y para la velocidad de la CPU será “hw.cpufrequency”. Se pueden ver todos los tipos de información que se puede consultar si en una terminal se usa “$ > sysctl -a”.

3) `LINUX` [20]. Si el sistema operativo en el que se ejecuta el juego es compatible con Linux entonces se lee la información de la CPU del comando *lscpu*. Para realizar esto primero hay que crear una *pipe*. A continuación se crea un proceso hijo con la función *fork*. Entonces los dos procesos están unidos mediante la *pipe* y se redirecciona la salida estándar del hijo hacia el extremo correspondiente de la *pipe* para que se pueda comunicar con el programa original. Después el hijo muta al programa *lscpu* con la función del sistema *exec*. Este realiza su función y escribe en la *pipe* la salida de su programa. El programa principal, después de crear al hijo lee de la *pipe* a la espera de que *lscpu* acabe de transmitirle todos los datos (recibe un EOF, *End Of File*). Finalmente, parsea la información recibida buscando los valores que interesan. Estos valores son los que vienen después de las palabras “CPU(s):” y “CPU MHz:”.

Hay multitud de sistemas operativos distintos con muchas versiones algunos, pero todos se pueden incluir en uno de los 3 tipos explicados. Si alguna de las APIs (*Application Programming Interface*) de estos sistemas operativos cambiase, se debería proporcionar soporte y actualizar el proyecto para adaptarse a las nuevas especificaciones del sistema.

6. Evaluación

La evaluación de este sistema consiste en la comprobación de que es capaz de formar partidas correctamente con los jugadores que se van conectando al servidor. Como no se dispone de ningún tipo de *testing* automatizado hay que realizar las pruebas a mano. Para ayudar a evaluar el sistema se han puesto *logs* (texto impreso en la terminal desde la que se ejecuta el sistema) que ayudan a identificar que es lo que está sucediendo y el punto en el que se encuentra la ejecución.

Obviamente, para comprobar que el sistema funciona se deberá tener como mínimo dos jugadores y el servidor activos. Esta será la configuración que se usará para realizar todos los tipos de pruebas. Además, como no se dispone de ningún servidor real, este estará en la misma máquina, junto con los dos jugadores. Esto hace que no se pueda dar el caso de que se pierda algún mensaje debido a la red. Hasta que no se disponga de los recursos necesarios para tener un servidor activo en una dirección IP accesible de Internet no se podrá comprobar que el reenvío de mensajes funciona correctamente.

El objetivo de las pruebas es ver que los dos jugadores se encuentran en la misma partida, aunque sea en la misma máquina. Esto será en el caso en que la prueba diseñada tenga como objetivo acabar en éxito. Pero no siempre las conexiones se comportarán como es de esperar.

Las pruebas tienen que tener en cuenta distintas situaciones posibles, incluso aquellos casos en los que se pueda caer alguno de los clientes (por fallo de la red o por voluntad propia). Para esto se realizarán pruebas en las que se desconectará a los clientes en cualquier punto del proceso de comunicación para ver que el sistema responde correctamente. En la mayoría de los casos, la respuesta esperada será el *timeout* de alguno de los clientes. Esto se ha hecho así debido a la falta de tiempo y recursos para implementar una solución más eficaz y que no lleve tanto tiempo. Lo que no debe ocurrir en ningún caso es que se cuelgue el servidor.

Una vez explicado cuál es el objetivo que se busca en las pruebas queda por detallar cómo se van a llevar a cabo. Para realizarlas hemos tenido en cuenta dos escenarios. El primero se lleva a cabo con los dos jugadores y el servidor en la misma

máquina, usando la dirección de *loopback*. En el segundo caso, también se usa el mismo ordenador, pero cada jugador se encontrará en sistemas operativos virtualizados mediante *Virtual Box* y el servidor en el sistema operativo nativo del ordenador. En este caso la dirección usada no será la de *loopback*, sino la que sea que tenga asignada la máquina virtual que ejecuta el servidor del sistema.

6.1. *Testing* local

Este es el tipo de *testing* que no usará máquinas virtuales. Para iniciar el sistema y poder comenzar las pruebas se deben seguir los pasos que se detallan a continuación. Para empezar, si no se dispone de un ejecutable del juego, hay que realizar el *build* (construcción) del programa.

Para crear el ejecutable primero se tiene que usar el *script* "bootstrap.sh", que se encarga de ejecutar los distintos programas del paquete *Autotools* con el objetivo de preparar el *build* para el sistema operativo del ordenador en que se está ejecutando. Este *script* genera el ejecutable *configure*, que comprueba que el ordenador disponga de todas las librerías y paquetes necesarios para proceder. También prepara los *Makefiles* para que se haga el *build* de las partes que se quiera del sistema, en caso de que no se vaya a utilizar alguna de ellas. Para ejecutarlo se usan los siguientes parámetros de configuración:

```
./configure --enable-master CODELEVEL=2 DEBUGLEVEL=3
```

Con la opción de `--enable-master` se indica que se quiere hacer *build* del servidor del sistema, además del juego. Por defecto sólo se hace el *build* del juego, así que está opción se debe especificar. `CODELEVEL` y `DEBUGLEVEL` son las opciones para personalizar el nivel de *warnings* y el tipo de *debug checks* que se quiere que realice el código del juego.

Además, si el *build* se realiza en un sistema MacOS se deben añadir unos *flags* extra (opciones para indicar dónde se encuentran ciertas librerías compartidas) para que el compilador pueda localizar las librerías de OpenGL y Glut. Estos *flags*, en el ordenador sobre el que se ha realizado el proyecto y el *testing* del sistema, son `LIBS="-framework OpenGL -framework GLUT"`.

Una vez se ha ejecutado *configure* entonces se puede pasar a ejecutar el *build* del sistema con *Make*. Para ello hay que ejecutar el comando `sudo make && make install`. Con esto se compila el código del juego y el servidor y se instala en el sistema (para esto último se necesita el *sudo*, ya que hay que escribir en carpetas del sistema operativo para las cuales se necesitan permisos de administrador).

Por último, para facilitar las cosas se puede crear un *soft link* al ejectuable que se encuentra en la carpeta *src/*, dentro de la carpeta */bin*, para poder ejecutarlo desde terminal sin necesidad de estar dentro de la carpeta del juego.

Ahora que ya están todo el sistema y juego compilados se pueden abrir 2 clientes y un servidor desde distintas terminales para empezar a realizar las pruebas. Para poder usar la opción de buscar partida mediante el sistema desarrollado hay que usar la opción “Quickplay”. Esta se encuentra en *Menu principal* > *Play Game* > *Multiplayer* > *Quickplay*. En la Figura 9 se puede ver un ejemplo de una de las pruebas realizadas.

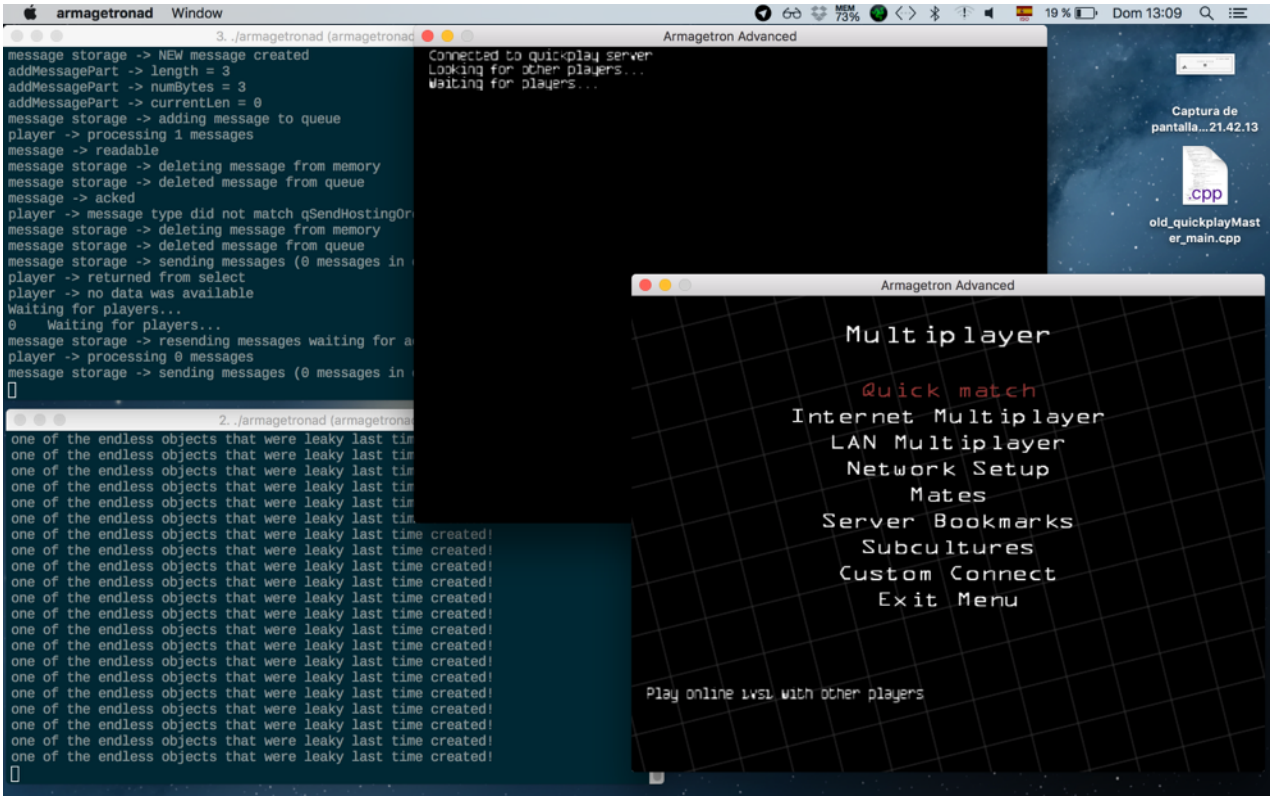

Figura 8. Captura de imagen tomada durante la realización de uno de los *tests* realizados

6.2. *Testing* con máquinas virtuales

En el caso de usar *Virtual Box* [21] para testear el sistema con máquinas virtuales el proceso es bastante similar. Para empezar hay que realizar el *build* del juego en cada máquina virtual, siguiendo los pasos descritos en el apartado anterior, para que se adapte al nuevo sistema operativo. El servidor se ejecutará en el sistema operativo nativo, que estará en la misma red que las dos máquinas virtuales creadas para los jugadores.

Para ejecutar los juegos se usará una imagen de un sistema operativo Debian en su última versión estable 8.6. En la Figura 10 se puede ver una imagen del software utilizado en las pruebas. Una vez estén las 2 máquinas virtuales en marcha, se inicia un juego en cada una de ellas de la misma manera que en el apartado anterior. Se inicia el servidor en el sistema nativo y entonces se accede a la opción “Quickplay” del menú para iniciar las pruebas, que son las mismas que se han realizado en el apartado anterior (comprobar el correcto funcionamiento del sistema, incluso simulando algunos errores de conexión).

7. Planificacion final y costes

Al inicio de este proyecto se planificó una duración estimada de 4 meses, empezando el 15 de febrero para acabar el 15 de junio. La estimación de la carga de trabajo que acarrearía era de unas 450 horas, aproximadamente. La realización de tareas se ha mantenido secuencial y en el mismo orden que se planeó inicialmente, pero ha habido ciertos cambios en la duración del proyecto que se detallan a continuación.

7.1. Desviaciones respecto el plan inicial

Uno de los cambios más importantes que ha tenido lugar es el añadido de una tarea para estudiar el código del juego, ya que su falta de documentación ha provocado problemas para entenderlo y ser capaz de adaptar el sistema para que funcione. Esto retrasó el proyecto más de lo previsto. Esta tarea ha tenido asignada una duración de 15 días. También se han añadido 7 días más para la redacción de la memoria, ya que ha sido más largo de lo esperado inicialmente.

Tal y como se mencionó en la descripción del proyecto, se eliminó la tarea de tomar medidas de rendimiento. Lo único medible del sistema desarrollado sería el tiempo que se tarda en encontrar partida, pero el efecto del código en esto es despreciable teniendo en cuenta que el factor humano implicado acarrea mucho más tiempo. El *ping* entre servidor y clientes también depende en gran medida de la localización de estos y su conexión a Internet, así que tampoco tenía sentido medirlo. A parte, no se dispone de un servidor y las pruebas se hacen en local, así que la mayoría de datos recopilados no tendrían ningún significado importante.

Esto ha conllevado una reducción de 10 días en el tiempo de planificación inicial, que son los que correspondían a la tarea de tomar medidas de rendimiento. La diferencia, junto con los 15 días añadidos debido a la nueva tarea y los 7 de la redacción de la memoria, es de 12 días más de lo previsto, que se traducen en un aumento en los costes directos e indirectos del proyecto.

Otro cambio importante en la planificación a mencionar es el retraso de la fecha de entrega original. Con el plan inicial se predecía que el proyecto finalizaría en Junio, pero debido a causas externas no pudo ser finalizado a tiempo y se ha tenido que prolongar hasta Octubre. Esto no afecta en la planificación del tiempo que se debía dedicar al proyecto, ya que si se ha retrasado la fecha de entrega ha sido por la falta de tiempo para dedicar al proyecto en Abril, Mayo y Junio (debido a otras asignaturas de la universidad), que ha sido compensado en los meses de Agosto, Setiembre y Octubre. En el diagrama de Gantt que se muestra en el siguiente apartado se puede ver que lo único que ha provocado esto es el desplazamiento a la derecha de la planificación, sin alterar nada más.

7.2. Planificación temporal final

A continuación se puede observar el diagrama de Gantt correspondiente a la planificación temporal final del proyecto, una vez este ha acabado (Figuras 11 a 13).

	Nombre	Duración	Inicio	Fin
1	1. Inicio del proyecto	11d	15/02/2016	29/02/2016
2	1.1. Búsqueda y puesta en marcha	2d	15/02/2016	16/02/2016
3	1.2. Aprendizaje	9d	17/02/2016	29/02/2016
4	2. Gestión del proyecto	30d	01/03/2016	11/04/2016
5	2.1. Definición del alcance	5d	01/03/2016	07/03/2016
6	2.2. Planificación temporal	5d	08/03/2016	14/03/2016
7	2.3. Gestión económica y sostenible	5d	15/03/2016	21/03/2016
8	2.4. Presentación preliminar	5d	22/03/2016	28/03/2016
9	2.5. Pliego de condiciones	5d	29/03/2016	04/04/2016
10	2.6. Presentación oral y documento final	5d	05/04/2016	11/04/2016
11	3. Estudio del código del juego	15d	12/04/2016	02/05/2016
12	4. Servidor	25d	12/07/2016	15/08/2016
13	4.1. Requisitos	1d	12/07/2016	12/07/2016
14	4.2. Diseño	2d	13/07/2016	14/07/2016
15	4.3. Implementación	22d	15/07/2016	15/08/2016
16	5. Modelo de conexión	25d	16/08/2016	19/09/2016
17	5.1. Requisitos	1d	16/08/2016	16/08/2016
18	5.2. Diseño	2d	17/08/2016	18/08/2016
19	5.3. Implementación	22d	19/08/2016	19/09/2016
20	7. Docs	14d	20/09/2016	07/10/2016
21	7.1. Redacción	12d	20/09/2016	05/10/2016
22	7.2. Defensa	2d	06/10/2016	07/10/2016

Figura 9. Tareas del diagrama de Gantt final

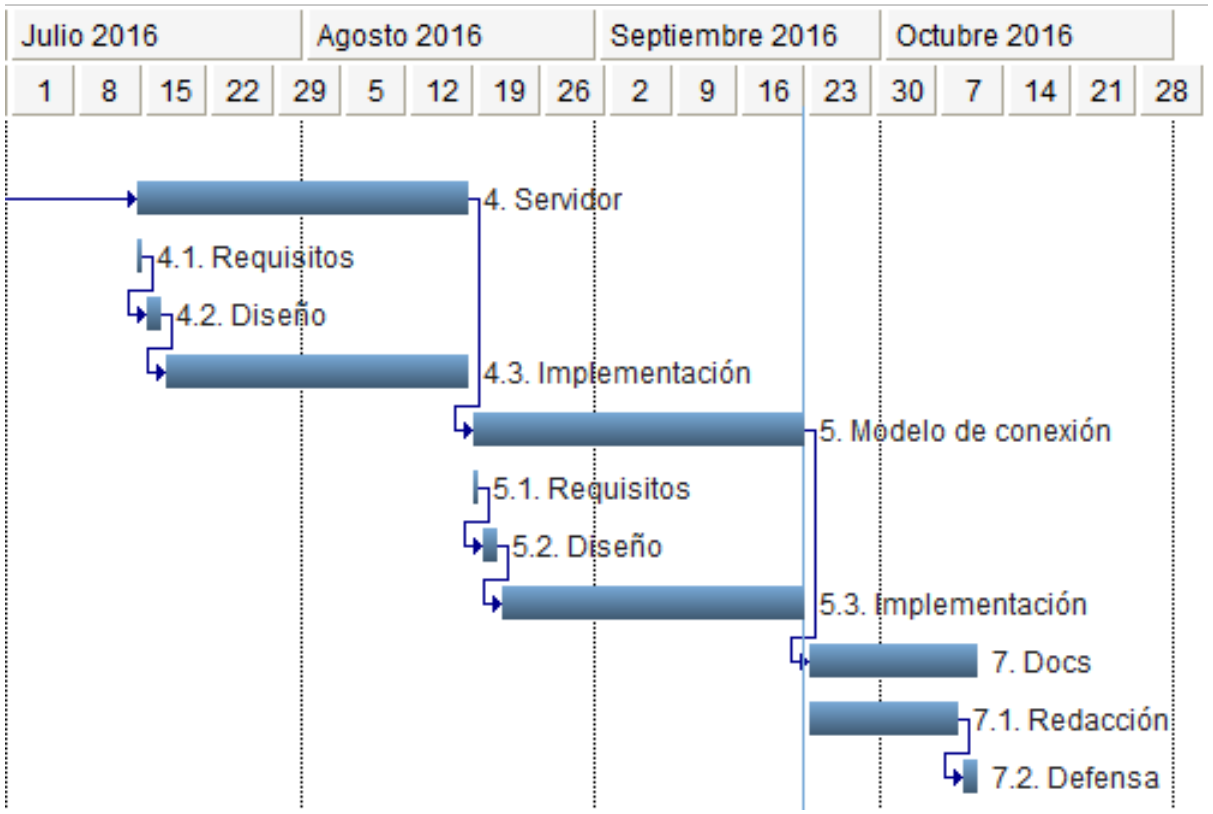

Figura 10. Diagrama de Gantt final de Febrero a Junio

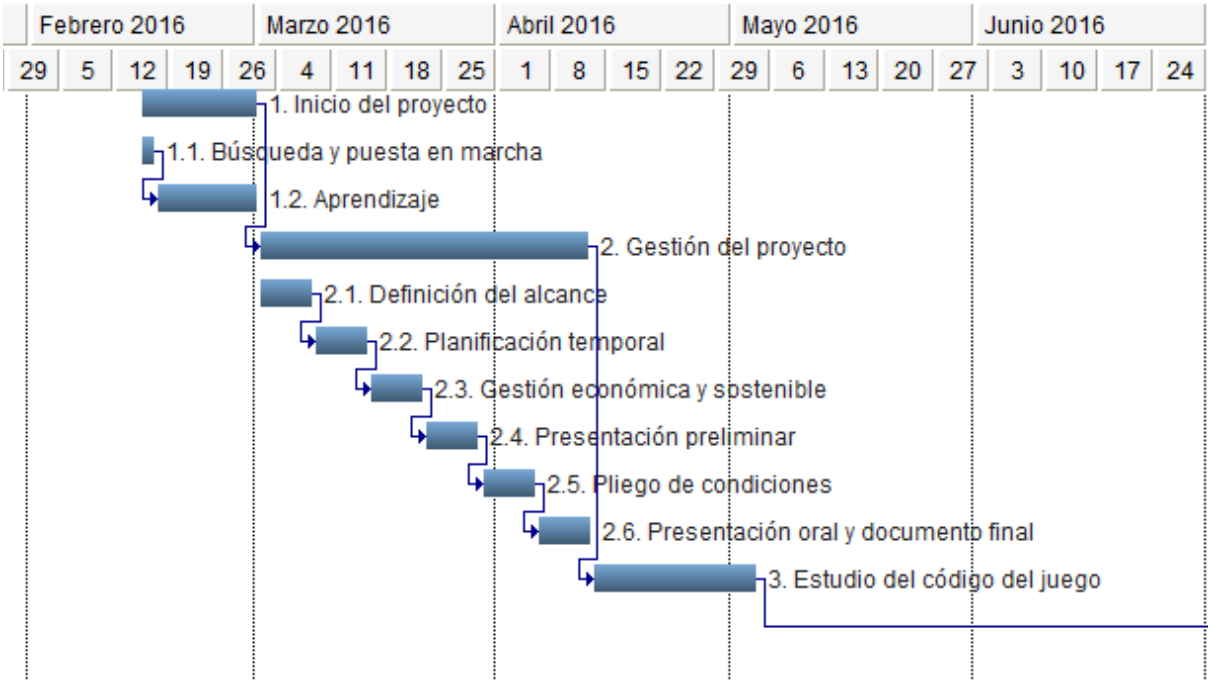

Figura 11. Diagrama de Gantt final de Julio a Octubre

7.3. Costes finales

En los costes finales se deben tener en cuenta los 12 días a tiempo parcial del desarrollador añadidos, así como el retraso de la entrega del proyecto. El primero tiene efecto sobre el coste directo del proyecto, ya que hay que pagar al trabajador. El segundo, en cambio afecta a los costes indirectos (transporte, alquiler, etc.).

Los coste directos se verán aumentados según la siguiente fórmula sencilla: 12 días * 4 horas / día * 8 € / hora. El resultado es de 384 euros que hay que añadir a los costes directos de personal.

Los costes indirectos son algo más complicados de calcular. Siguiendo la tabla que se encuentra en el punto 2.6, bajo “Costes generales indirectos”, hay que tener en cuenta el coste añadido de Internet, local, agua, luz, gas y transporte durante 3 meses más. Julio no se tiene en cuenta ya que el proyecto estuvo parado durante ese mes, tal y como se puede comprobar en el diagrama de Gantt del apartado anterior. Usando los precios de esa misma tabla y multiplicando por 3 meses se obtiene un resultado de 1.548 euros.

Sumando los costes directos e indirectos añadidos, el coste del proyecto ha aumentado en 1.932 euros. Teniendo en cuenta que se habían destinado 1453.7245 euros entre costes de contingencia y de imprevistos al proyecto, esto supondría un coste adicional respecto al presupuesto inicial de 478.2755 euros.

8. Análisis de sostenibilidad

En este capítulo se analiza la sostenibilidad del proyecto. La sostenibilidad es un término que se refiere al impacto que tiene un proyecto sobre la economía, la sociedad o el medio ambiente. Este impacto puede ser negativo, en cuyo caso se estaría hablando de baja sostenibilidad, o positivo.

Para realizar este análisis se tendrá en cuenta el impacto del proyecto durante su producción y su vida útil. Además también se considerarán posibles riesgos que afecten negativamente a la sostenibilidad. Para hacer una valoración lo más objetiva posible se dará una puntuación a cada uno de los campos de la siguiente "Matriz de Sostenibilidad" (Tabla 9).

	Proyecto en producción	**Vida útil y resultados**	**Riesgos**
Medio ambiente	Análisis de recursos	Viabilidad económica	Impacto social
Puntuación	[0 - 10]	[0 - 10]	[0 - 10]
Economía	Huella ecológica	Coste final	Impacto social
Puntuación	[0 - 20]	[0 - 20]	[0 - 20]
Sociedad	Inconvenientes ambientales	Riesgos económicos	Inconvenientes sociales
Puntuación	[-20 - 0]	[-20 - 0]	[-20 - 0]

Tabla 9. Matriz de sostenibilidad con sus rangos de puntuaciones

En cada apartado siguiente de este capítulo se reflexionará sobre el impacto del proyecto durante sus etapas y sus riesgos. En el último apartado se encuentra esta misma matriz, con las puntuaciones asignadas según las reflexiones realizadas.

8.1. Medio ambiente

Actualmente, el sistema de comunicación del juego online que se ha modificado es costoso desde el punto de vista ambiental, ya que requiere de unos servidores activos que mantengan las partidas. El juego no tiene mucho caudal de jugadores, por lo que estos servidores no son demasiados, pero si se diese el caso de que se pusiera de moda y su número de jugadores aumentase, entonces podría llegar a ser un problema en cuanto a emisiones de CO2 debido al número de ordenadores necesarios para atender a todos los clientes.

Con la mejora propuesta en este trabajo todas estas emisiones y consumo de energía extra de los servidores del juego se reducirá prácticamente a 0. Lo único que hará falta será el servidor del sistema de este proyecto para atender a los jugadores que busquen partida y a partir de entonces serán los propios clientes los que hagan a la vez de jugador y servidor. No sólo se gana en energía y en emisiones de CO2, sino también en reutilización de recursos, ya que se aprovechan ordenadores que ya están en funcionamiento para realizar más tareas y así evitar tener que usar recursos adicionales que no son necesarios.

Durante el desarrollo del proyecto sólo se utiliza un ordenador portátil, que ya se utilizaba anteriormente, por lo que no se gasta en recursos adicionales y tanto el consumo de energía como las emisiones de CO2 correspondientes a la realización del proyecto son despreciables.

Finalmente, no hay ningún riesgo que pueda aumentar la huella ecológica de este sistema, ya que está diseñado precisamente para disminuirla lo máximo posible.

8.2. Economía

Para considerar si un proyecto es viable económicamente es necesario realizar un estudio de costes como el que se ha hecho en los capítulos 2 y 7 de este documento. Como se puede ver, la mitad de los costes del proyecto aproximadamente son de recursos humanos; es lo que vendría a ser el tiempo dedicado al proyecto del trabajador. El resto son gastos inevitables que se deberían pagar incluso aunque no se

estuviese desarrollando ningún tipo de software, por lo que el proyecto es bastante viable económicamente debido a su bajo coste de producción.

El punto fuerte de este proyecto radica en el ahorro económico que supondría para un potencial cliente su adopción. Actualmente, si una empresa tuviese servidores para proporcionar partidas del videojuego en cuestión a clientes, entonces cada “X” clientes nuevos tendría que gastar dinero para comprar un nuevo servidor/máquina que les pudiese dar servicio. No sólo necesitaría un coste inicial de inversión, sino también el coste de mantenimiento. Con la adopción de este sistema el número de servidores necesario no dependería del número de jugadores, sino que sería fijo e igual a 1. Este hecho facilitaría y también reduciría el coste en personal y en energía de la empresa enormemente. Estas consideraciones muestran que el sistema desarrollado puede ser perfectamente competitivo.

El código del proyecto para el servidor es portable a cualquier sistema UNIX, aunque si se quisiera usar en un sistema WINDOWS habría que hacer trabajo adicional para asegurar su funcionamiento correcto. El código que se inserta en el cliente, una vez finalizada la API será perfectamente portable a cualquier sistema, por lo que no requiere de ninguna tarea de adaptación. Además, el código desarrollado únicamente necesita mantenimiento si un cliente en particular desea añadir alguna funcionalidad extra al servidor o cliente.

El tiempo dedicado a cada tarea del proyecto ha sido el mínimo posible, ya que se quiere tener el coste lo más bajo posible. Si se quisiera realizar el proyecto en menos tiempo entonces se habrían necesitado más recursos humanos y materiales. Además, cada tarea ha sido pensada para desarrollar partes vitales del funcionamiento del sistema, ya que no se encuentran en Internet otros proyectos de código abierto que se puedan aprovechar para realizar la tarea requerida (quizá alguna empresa tiene alguna implementación parecida, pero es privada y no accesible).

En cuanto a riesgos, se debe tener en cuenta que al ser un proyecto Open Source y libre no pretende sacar mucho provecho económico. Esto hace que el riesgo económico sea altísimo, ya que no se planea recuperar el dinero invertido para llevar a cabo el desarrollo del proyecto. Los clientes serían empresas que quisieran comprar una licencia que les permitiese seguir desarrollando el sistema de forma privada.

8.3. Sociedad

El aspecto social de este proyecto es el menos destacado, ya que no hay nadie a quien se le mejore socialmente su calidad de vida ni existe una necesidad real para su desarrollo, a parte de la económica o ambiental. Los jugadores del juego sobre el que se ha implementado el sistema seguirán jugando sin darse cuenta de que algo ha cambiado, excepto la nueva opción del menú. Además es un proyecto incluido en el sector de los videojuegos, que sólo tiene lugar en aquellos países y para aquellas personas cuya situación social es suficientemente buena como para poder permitirse este entretenimiento (se supone que si alguien juega a videojuegos es porque tiene todas sus necesidades básicas cubiertas).

Los únicos que verían una pequeña mejora son aquellos miembros de la comunidad del juego que tienen algún servidor público con partidas para jugar. Al ser un juego Open Source, es la comunidad de jugadores la que se encarga de que haya servidores que permitan jugar a todo el mundo (ya que no hay ninguna empresa que se encargue). Estos verían algo aliviada su responsabilidad, ya que no habría necesidad de mantener tantas máquinas y eso supondría un pequeño ahorro de dinero y de problemas para ellos.

En cuanto a riesgos, no puede haberlos ya que el impacto social del proyecto es casi inexistente. Esto significa que no habrá nadie a quien perjudique. Tampoco creará ningún tipo de dependencia sobre la población ya que al ser software libre cualquiera puede continuar con el trabajo realizado o modificarlo a su antojo.

8.4. Evaluación de la sostenibilidad

Finalmente, en este apartado se presenta la Matriz de Sostenibilidad con sus puntuaciones asignadas, basadas en lo comentado en los apartados anteriores. El valor total refleja lo sostenible que es el proyecto. Los impactos que tiene el proyecto ambientalmente, económicamente y socialmente durante su producción y a lo largo de su vida útil son positivos (por eso sus puntuaciones suman). En cambio los posibles riesgos a los que se enfrenta el proyecto son negativos, por lo que sus puntuaciones restan sostenibilidad al proyecto.

La suma de todos los valores de la Matriz de Sostenibilidad indica cómo de sostenible es el proyecto, con un rango de valores que va desde -60 (siendo entonces un proyecto nada sostenible) a 90, que representaría un proyecto perfectamente sostenible en todas sus facetas. A continuación se encuentra la Matriz de Sostenibilidad con las puntuaciones asignadas (Tabla 10).

	Proyecto en producción	**Vida útil y resultados**	**Riesgos**
Medio ambiente	Análisis de recursos	Huella ecológica	Inconvenientes ambientales
Puntuación	9 [0 - 10]	15 [0 - 20]	0 [-20 - 0]
Economía	Viabilidad económica	Coste final	Riesgos económicos
Puntuación	6 [0 - 10]	18 [0 - 20]	-15 [-20 - 0]
Sociedad	Impacto social	Impacto social	Inconvenientes sociales
Puntuación	0 [0 - 10]	2 [0 - 20]	0 [-20 - 0]

Tabla 10. Matriz de sostenibilidad con puntuaciones

Sumando los valores de cada celda se obtiene un valor de sostenibilidad de 35. Esto es una sostenibilidad bastante positiva teniendo en cuenta que se trata de un proyecto realizado durante un Trabajo Final de Grado.

9. Conclusiones

Con este proyecto se pretendía crear un sistema capaz de integrarse en cualquier videojuego que sirviera para juntar a gente en una misma partida a través de Internet. Para ello se ha usado como prueba el videojuego *Armagetron Advanced*.

El sistema de partidas online de la mayoría de juegos de hoy en día sigue un modelo de conexión cliente/servidor en el cual el servidor forma parte de la empresa que desarrolla el juego. Con el proyecto se pretendía ofrecer una solución sencilla para todos aquellos juegos que quisieran usarlo, ya que el sistema desarrollado no requiere de una infraestructura de servidores debido al nuevo modelo de conexión que tiene implementado.

Mirando hasta dónde se ha llegado en el desarrollo del sistema se puede ver que no se ha conseguido exactamente esto. El sistema funciona perfectamente con el juego en el que se ha integrado, pero aún le queda mucho por delante para tratarse de algo que pueda ser usado por cualquier juego. Hay que abstraer más el código del sistema para que pueda ser integrado en cualquier otro juego porque ahora mismo no tiene una API que poder usar fácilmente.

Las dos grandes dificultades al realizar este proyecto han sido el protocolo de mensajes entre los clientes y el servidor e intentar usar el código del juego para integrar el sistema. La primera es debido a la dificultad que conlleva realizar programas que se comuniquen a través de la red, con tantas posibilidades abiertas y cosas que pueden salir mal; hay que tener en cuenta cualquier posible fallo, tanto propio como externo. Es posible que alguno haya escapado a las pruebas de evaluación del sistema. El segundo ha sido debido a la mala documentación del código del juego, cosa que muestra la importancia de ser capaz de hacer que los demás sean capaces de entender que es lo que uno está programando.

Aunque, a pesar de las dificultades que han surgido, con esfuerzo y perseverancia se ha logrado tener un sistema con el funcionamiento esperado en el videojuego mencionado. Teniendo en cuenta los conocimientos previos en el lenguaje usado y en redes, se ha necesitado de mucho trabajo para llegar a tener este software en funcionamiento

9.1. Trabajo futuro

Este proyecto aún está inmaduro si pretende llegar a ser usado por un amplio público del sector de videojuegos y hay que ser consciente de ello. A continuación se encuentran las cosas que creo que son más relevantes y que no tiene.

La seguridad es algo muy importante y de la cual no se ha tocado nada en este trabajo. Habría que asegurarse de que el servidor se encuentra seguro frente a posibles ataques de DDOS que pueda sufrir. No soy un experto en seguridad informática, pero seguro que este no es el único peligro frente al que habría que proteger el servidor.

Como se ha comentado previamente en este mismo capítulo, falta por crear una API que pueda ser usada fácilmente para integrar el sistema en los juegos. Los métodos dependen demasiado del código del juego en cuestión. Habría que lograr que fueran más independientes y más flexibles en cuanto a las funciones que pueden ser capaces de realizar, para que así puedan satisfacer las necesidades de cualquiera que pretenda usar el sistema.

Una propiedad que sería bueno tener en cuenta para escoger quién hará de servidor, y que por falta de tiempo no se ha podido implementar, es el *ping* entre los distintos clientes. Para esto habría que cambiar parte del protocolo de comunicación entre el servidor y los clientes. Después de haber escogido los participantes de una partida, el servidor debería enviarles un mensaje con la información de todos los jugadores a cada uno. Entonces cada jugador debería comprobar el *ping* que tiene con todos los participantes, calcular la media y enviar la información de vuelta al servidor. Este recogería la media de *ping* de cada jugador de la partida y tendría en cuenta este valor, junto con la información de la CPU, para escoger el jugador que debería crear la partida.

Finalmente, para intentar evitar las trampas se podría tratar de mejorar el modelo de tal forma que el que creará la partida y hiciese de servidor fuera otro jugador de una partida distinta. De esta manera se conseguiría el mismo efecto que con el modelo cliente/servidor tradicional, pero sin la necesidad de los servidores necesarios para ello. Por otro lado, esto traería un montón de nuevos problemas a solucionar, como por ejemplo, la migración de la partida a un nuevo ordenador en caso de que el jugador que la mantiene se desconecte.

9.2. Conclusiones personales

Personalmente, creo que este proyecto se me ha quedado demasiado grande. Al principio tenía un montón de ideas que quería implementar, pero luego he visto lo complicado que era todo y lo mucho que costaría realmente hacer lo que tenía en mente.

Alejandro y Javier, los directores del proyecto, me ayudaron a pulir la idea inicial hasta convertirla en este trabajo. Gracias a ellos, y a todo lo que he tenido que investigar para poder escribir esta memoria, he aprendido muchas cosas sobre la manera en la que los videojuegos online funcionan. También he practicado escribiendo código para llevar a cabo la parte práctica de este trabajo.

Desarrollar este proyecto me ha servido para aprender muchas cosas que no me habían enseñado en la carrera. Esto es algo bueno, pero no siempre ha sido agradable. Al empezar casi no recordaba nada de C++ y se me hizo muy cuesta arriba el proyecto. Tuve que leer mucho para aprender a usar las herramientas que necesitaba, ya que apenas entendía el código del juego que tenía que modificar.

Usar código de un programa de software libre me ha hecho ver la importancia de documentar bien el código, cosa que tiene su gracia porque es algo que el sistema creado también tiene pendiente. Entiendo que es algo importante pero que nunca va a recibir la atención que merece, ya que lo más importante en un proyecto es que este funcione bien y no que pueda ser entendido fácilmente por otros.

Finalmente, creo que llevar a cabo este trabajo ha valido la pena. Las motivaciones que me llevaron a intentar crear este sistema de software han sido satisfechas, ya que he acabado teniendo unos conocimientos que considero más que suficientes para lo que me había propuesto. Sólo falta por ver si en el futuro me ayuda a conseguir un puesto en alguna empresa de videojuegos.

10. Bibliografia

[1] Newzoo (21/04/2016). *The global games market reaches $99.6 billion in 2016, mobile generating 37%* - https://newzoo.com/insights/articles/global-games-market-reaches-99-6-billion-2016-mobile-generating-37/

[2] Newzoo (August, 2016). *Most watched games on Twitch* - https://newzoo.com/insights/rankings/top-games-twitch/

[3] Fiedler, G. (n.d.). *What every programmer needs to know about game networking* - http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/

[4] Terrano, M., Bettner, P. (22/03/2001). *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond* - http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php

[5] Unreal Engine (n.d.). *Client Server Model* - https://udn.epicgames.com/Three/ClientServerModel.html

[6] Freeman, R. E. (1984). *Strategic Management: A Stakeholder Approach*. Cambridge: Cambridge University Press

[7] Oxagile (05/02/2014). *Waterfall software development model* - http://www.oxagile.com/company/blog/the-waterfall-model/

[8] Atlassian (n.d.). *Git tutorials: Git Workflows* - https://www.atlassian.com/pt/git/workflows#!workflow-gitflow

[9] Gantter (n.d.). *Collaborative cloud scheduling made easy* - http://gantter.com/

[10] JobTonic (n.d.) *Buscar trabajo en todos los sitios de empleo!* - http://www.jobtonic.es/

[11] SóloIngeniería.Net (n.d.). *Cuanto me costará imprimir el PFC?* - http://www.soloingenieria.net/foros/viewtopic.php?f=2&t=25679

[12] Armagetron Advanced (n.d.). *A Tron Clone in 3D* - http://www.armagetronad.org/

[13] Cplusplus (n.d.). *A brief description* - http://www.cplusplus.com/info/description/

[14] Gary V. Vaughan, Ben Elliston, Tom Tromey, Lance Taylor (October 2000). *GNU Autoconf, Automake and Libtool*. New Riders

[15] Scott Chacon, Ben Straub (2014). *Pro Git*. Apress

[16] Open Group, IEEE (2004). *Standards Information Base* - http://pubs.opengroup.org/onlinepubs/009695399/functions/recv.html

[17] Michael Kerrisk (10/08/2016). *Linux programmer's manual* - http://man7.org/linux/man-pages/man2/select.2.html

[18] Microsoft Developer Resources (2016). *Getting Hardware Information* - https://msdn.microsoft.com/en-us/library/windows/desktop/ms724423(v=vs.85).aspx

[19] OS X Man Pages (2016). *BSD Library Functions Manual: SYSCTL* - https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man3/sysctl.3.html#//apple%5Fref/doc/man/3/sysctl

[20] Cai Qian, Karel Zak, Heiko Carstens (n.d.). *lscpu - Linux man page* - https://linux.die.net/man/1/lscpu

[21] Oracle (n.d.). *Virtual Box* - https://www.virtualbox.org/

10.1. Referencias

Como referencias se incluyen 2 libros que he leído casi por completo y consultado en múltiples ocasiones y que me han ayudado a realizar este proyecto. Estos han sido:

[1] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo (Agosto, 2012). *C++ Primer, Fifth Edition*. Addison-Wesley

[2] Brian “Beej Jorgensen” Hall (11/03/2016). *Beej’s Guide to Network Programming. Using Internet Sockets*. Sin editorial.

10.2. Fuentes de las imágenes

En este último apartado de la bibliografía se incluyen las fuentes de las imágenes obtenidas a través de alguna página web y que no han sido tomadas o creadas por mi:

Figura 1. http://www.freelan.org/static/images/peer-to-peer.png

Figura 2. http://www.freelan.org/static/images/client-server.png

Figura 3. http://www.umsl.edu/~hugheyd/is6840/images/Waterfall_model.png

Figura 4. http://www.armagetronad.org/images/header.gif

Figura 5. http://www.armagetronad.org/screenshots/screenshot_23.png

A. Apéndice

En este Apéndice se pueden ver las cabeceras (en C++) de las principales clases usadas en el código del sistema, de las cuales se habla en algunos capítulos de la memoria.

A.1. qMessage.h

```
/*
 * This class represents a message.
 */
class qMessage {
     private:
          uchar *buffer;
          ushort messLen;
          ushort currentLen;
          uchar type;

     public:
          qMessage();
          explicit qMessage(uchar type);
          virtual ~qMessage();

          // Getters & Setters
          inline uchar *getBuffer() { return buffer; }
          inline ushort getMessageLength() { return messLen; }
          inline ushort getCurrentLength() { return currentLen; }
          inline uchar getType() { return type; }
          inline void setMessageLength(ushort mLen) {
               messLen = mLen;
          }
          inline void setCurrentLength(ushort cLen) {
               currentLen = cLen;
          }

          ushort readShort(const uchar *buf, int &bytesRead);
          void writeShort(ushort val, int &position);
          void addMessagePart(const uchar *buf, int numShorts);
          bool isMessageReadable();
          void acknowledgeMessage(const MQ::iterator &it,
               qMessageStorage *ms);

          virtual void handleMessage(const MQ::iterator &it,
               qMessageStorage *ms);
          virtual int send(int sock);
          virtual void prepareToSend();
};
```

A.2. qMessageStorage

```
typedef std::map<int, qMessage*> MQ;
typedef std::pair<int, qMessage*> messElem;


class qMessageStorage {
     private:
          // a message can only be in one queue at a time

          MQ receivedQueue;
          MQ sendingQueue;
          MQ pendingAckQueue;

          int getSocketReferences(const MQ::iterator &it);

     public:
          qMessageStorage() {}
          virtual ~qMessageStorage();

          MQ::iterator getMessageFromQueue(int sock, MQ &queue);
          inline MQ &getReceivedQueue() { return receivedQueue; }
          inline MQ &getSendingQueue() { return sendingQueue; }
          inline MQ &getPendingAckQueue() {
               return pendingAckQueue;
          }

          void deleteAllMessages(int sock);
          void deleteMessage(MQ::iterator &it, MQ &queue);
          void deleteMessage(int sock, MQ &queue);
          void addMessage(const messElem &elem, MQ &queue);
          void moveMessage(MQ::iterator &it, MQ &originQueue,
               MQ &destQueue);
          void handleData(const uchar *buf, int numBytes,
               int sock);

          qMessage *createMessage(uchar type);
          virtual void processMessages() = 0;
          virtual void sendMessages();
          virtual void resendUnacked();
};
```

A.3. qServer & qServerInstance

```
typedef std::map<int, qPlayer*> PQ;
typedef std::multimap<uint, qPlayer*> MatchQ;


/*
 * This class provides all common methods to create a server.
 * It creates a server capable of listening and answering
 * to different connections and relies in the handle data
 * method of its derived classes to perform each different
 * task needed.
 */
class qServer {
     private:
          int listener;
          int fdmax;
          fd_set master;

     public:
          qServer();
          virtual ~qServer();

          // getters
          inline fd_set *getFileDescriptorList() {
               return &master;
          }
          inline int *getFDmax() { return &fdmax; }
          inline int *getListener() { return &listener; }

          virtual void getData() = 0;
};


/*
 * Class used to implement our quickplay server.
 * The handle data will listen for the requests of the different
 * players and queue them or assign them to their corresponding
 * match.
 */
class qServerInstance : public qServer, public qMessageStorage {
     private:
          static uint matchesIds;

          PQ playerQueue;
          MatchQ matchesQueue;

          void fillClientPlayers(
               std::vector<qPlayer *> &cPlayers, uint matchId);
          void sendConnectMessages(
               const std::vector<qPlayer *> &cPlayers);

     public:
          qServerInstance();
          ~qServerInstance();
```

```
        inline const PQ &getPlayerQueue() {
             return playerQueue;
        }
        inline uint getNextMatchId() {
             return matchesIds++;
        }
        inline void addPlayer(qPlayer *&newPlayer, int sock) {
             playerQueue.insert(std::pair<int, qPlayer*>
                  (sock, newPlayer));
        }
        inline void deletePlayer(int sock) {
             PQ::iterator it = playerQueue.find(sock);
             playerQueue.erase(it);
        }

        qPlayer *getPlayer(int sock);

        void deletePlayerFromMatches(int sock, uint id);
        void addPlayerToMatches(qPlayer *player, uint id);
        void getData();
        void processMessages();
        int prepareMatch();
};
```

A.4. qConnection & qPlayer

```
class qConnection {
     private:
          int sock;
          struct sockaddr_storage remoteaddr;

     public:
          qConnection();
          qConnection(int socket,sockaddr_storage remoteaddress);
          virtual ~qConnection();

          inline int getSock() { return sock; }
          inline int *getSockAddr() { return &sock; }
          inline sockaddr_storage *getSockaddrStorage() {
               return &remoteaddr;
          }
          inline bool active() { return sock >= 0; }
          inline void closeConnection() {
               close(sock); sock = -1;
          }
};


class qPlayer : public qConnection, public qMessageStorage {
     private:
          PlayerInfo info;
          uint matchId;
          bool master;

     public:
          qPlayer();
          qPlayer(int socket, sockaddr_storage remoteaddress);
          ~qPlayer();

          inline uint getMatchId() { return matchId; }
          inline bool isMaster() { return master; }
          inline bool isPlayerAvailable() {
               return (matchId == 0) && info.isInitialized();
          }
          inline bool gameFound() {
               return master || getSockaddrStorage()->ss_family;
          }

          void setMatchId(uint id) { matchId = id; }
          void setMaster(bool m) { master = m; }
          void setInfo(qPlayerInfoMessage *message);
          void setConnection(qSendConnectInfo *message);
          void sendMyInfoToServer();
          void sendMatchReadyToServer();
          void processMessages();
          bool ackReceived();
          bool hasBetterPC(qPlayer *other);
          int getData();
};
```

A.5. PlayerInfo

```
class PlayerInfo {
     private:
          uchar numCores;
          uchar cpuSpeedInteger;
          uchar cpuSpeedFractional;

          int getCpuInfo(uchar *nCores, uchar *cpuSI,
               uchar *cpuSF);

     public:
          PlayerInfo();
          PlayerInfo(uchar numCores, uchar cpuSpeedInteger,
               uchar cpuSpeedFractional);

          // getters
          inline uchar getNumCores() const { return numCores; }
          inline uchar getCpuSpeedInt() const {
               return cpuSpeedInteger;
          }
          inline uchar getCpuSpeedFrac() const {
               return cpuSpeedFractional;
          }

          bool isInitialized() { return numCores > 0; }
          int setCpuInfo();
          void setProperties(uchar nCores, uchar cpuSI,
               uchar cpuSF, ushort p);
};
```